UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie électrique et de génie informatique

INTÉGRATION DE RÉSEAUX DE NEURONES À DÉCHARGE SUR UNE PLATEFORME ROBOTIQUE

Mémoire de maîtrise

Spécialité: génie électrique/informatique

Jean-Sébastien DESSUREAULT

Jury: François MICHAUD (directeur)

André CYR

Éric PLOURDE

Sherbrooke (Québec) Canada Mai 2019

DÉDICACE

À ma mère, Michelle Baril, qui a toujours cru en moi et qui a toujours été là pour moi.

RÉSUMÉ

S'inspirant de la biologie animale et humaine, les réseaux de neurones à décharge (RND) représentent une technologie prometteuse en intelligence artificielle. Les réseaux de neurones artificiels traditionnels mimiquent sommairement les réseaux de neurones biologiques. C'est parce qu'ils imitent mieux les neurones biologiques que les RND pourraient devenir de plus en plus intéressants, surtout que l'avènement de processeurs neuromorphes ou de cartes graphiques qui permettent le traitement en parallèle facilitent la mise en œuvre de RND.

Toutefois, l'absence d'outils facilitant l'implémentation de RND sur des robots rend difficile leur exploitation, ce qui limite la validation de leurs capacités pour améliorer l'intelligence des robots. Par exemple, les RND ne sont pas encore bien adaptés pour s'exécuter sur le système d'exploitation ROS, de loin l'environnement de développement robotique le plus exploité par la communauté scientifique. Ce mémoire propose un nouvel outil, ROS-SNN, qui intègre un RND sur la plateforme ROS[1]. Il décrit l'outil et présente ses conditions et contraintes de fonctionnement. ROS-SNN propose un format de déclaration standard de RND, ainsi qu'une série d'utilitaires qui permettent de bien convertir et visualiser les données sous forme de graphiques. Des tests dans différentes configurations ont permis d'identifier le temps d'exécution de cycle de RND sur un nano processeur, le Raspberry Pi 3B, permettant d'identifier les conditions pour assurer un traitement satisfaisant les contraintes de traitement temps réel pour une application donnée. La librairie ROS-SNN a aussi été validée sur une nouvelle plateforme robotique, nommée Spike, démontrant la faisabilité de déployer des RND sur un robot.

Mots-clés : Réseaux de neurones artificiels à décharge, robotique bio-inspirée, *Leaky Integrate and Fire*, *Robot Operating System* (ROS).

[1] L'outil présenté peut être téléchargé à partir de ce lien: https://github.com/jsdessureault/ros_snn

REMERCIEMENTS

Je tiens à remercier mon directeur François Michaud d'avoir cru en moi, malgré les conditions difficiles qu'implique le cumul de mon travail d'étudiant et d'enseignant à temps plein. Sa compétence et son appui à ma maîtrise justifient amplement mon choix d'étudier à l'Université de Sherbrooke, loin de chez moi.

Je veux remercier également Dominic Létourneau et André Cyr. Leur aide, leur soutien et leur patience me sont précieux.

Je tiens à souligner également d'appui indéfectible de ma mère Michelle, de mon ex conjointe Sylvie, de mes fils Yohann et Esteban, ainsi que de ma conjointe Julie. La noble cause de l'intelligence artificielle les prive souvent de leur fils ou de leur conjoint. Merci à vous.

Merci à mon défunt père, René-Paul, qui m'a transmis son constant désir d'apprendre, ainsi que sa curiosité.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LEXIQUE

Indigo Igloo	Version de ROS
Kinetic Kame	Version de ROS
Matplotlib	Librairie pour affichage de graphiques
Rosbag	Utilitaire ROS pour enregistrement de données à partir des senseurs
Topic	Outil ROS pour faciliter la communication entre les nœuds
Turtle Jade	Version de ROS
ARMHF	ARM *Hard Float*

LISTE DES ACRONYMES

GPU	*Graphics Processing Unit* : Processeur graphique.
I&F	*Integrate and Fire*: Modèle de réseau de neurones à décharge.
LI&F	*Leaky Integrate and Fire*: Modèle de réseau de neurones à décharge.
RND	Réseau de neurones à décharge.
ROS	*Robot Operating System* : Système d'exploitation utilisé pour la robotique.
SNN	*Spiking Neural Network*
STDP	*Spike Timing Dependent Plasticity* : Processus d'apprentissage qui permet d'ajuster les poids synaptiques entre les neurones d'un RND.

CHAPITRE 1 INTRODUCTION

Alan Turing a été l'un des premiers à jeter les bases de l'intelligence artificielle [1]. Les premières réflexions sur le sujet tendaient à répliquer l'intelligence humaine, sans pour autant imiter la façon dont le cerveau fonctionne. Turing, suivi de quelques autres scientifiques, se sont posés la question à savoir si une machine pouvait penser. Ensuite, ils se sont interrogés sur la façon d'évaluer cette nouvelle forme d'intelligence. Les premiers tests d'intelligence ont été créés. Le plus célèbre est le Test de Turing [1] qui consiste à tenter de différencier l'humain de de la machine lors d'un dialogue à l'aveugle avec un agent conversationnel.

Les travaux de Turing au sujet de l'intelligence sont cependant demeurés théoriques. Afin de mieux connaitre l'intelligence, il fallait améliorer nos connaissances sur le cerveau. À ce sujet, les expériences de Hodgkin et Huxley [2] sur les calmars géants ont permis de développer les premiers modèles en neurosciences computationnelles. Un physiologiste canadien, Donald Hebb [3], a également largement contribué à la compréhension du cerveau et de sa plasticité.

À partir de ces travaux sur le thème de la biologie du cerveau humain, McCulloch et Pitts [4] ont publié les premiers travaux au sujet de modèles mathématiques et logiques, mettant ainsi la table à la reproduction de certains principes d'inspiration biologique sur un support informatique. Le fameux Perceptron de Rosenblatt [5] fut le premier réseau de neurones artificiels à avoir la capacité d'apprendre de ses propres expériences. Hopfield et son « modèle de Hopfield » [6], tout comme Hinton et Sejnowski et leur Machine de Boltzmann [7], ont également contribué à la neuroscience computationnelle.

Les réseaux de neurones artificiels ont ensuite évolué pour créer un modèle plus spécifique de neurones biologiques : les réseaux de neurones à décharge (RND). Il existe plusieurs modèles de RND, dont ceux de Hodgkin et Huxley et Izhikevich [8] [9], mais probablement que le plus intéressant pour la robotique est le modèle *Leaky Integrate and Fire* (LI&F) [9]. Ce modèle très simple permet l'exécution d'un RND sur une plateforme robotique qui ne dispose typiquement que de peu de puissance de calculs.

Il n'existe aucun outil qui intègre parfaitement les RND et la robotique. Afin de créer des RND sur une plateforme robotique, il faut arriver à choisir une librairie parmi plusieurs, et

programmer un algorithme qui génère le RND en fonction du problème à résoudre, sans que l'algorithme soit ré-exploitable dans une autre problématique. Il faut donc ainsi réinventer la roue, pour chaque implémentation de RND sur une plateforme robotique.

Donc, ce projet de recherche se positionne à l'intersection des neurosciences computationnelles et de la robotique. Il vise à intégrer les RND à la plateforme robotique *Robot Operating System* (ROS) [10], la plus utilisée dans la communauté robotique, de façon paramétrable et réutilisable. Pour démontrer leur exploitabilité, le but est de développer une librairie de RND intégrée à ROS, nommé ROS-SNN (pour *Spiking Neural Network*), afin d'exploiter des RND sur des systèmes ordinés aussi simples qu'un nano processeur embarqué.

Le présent document est organisé de la façon suivante. Le chapitre 2 présente une revue de la littérature sur l'utilisation de RND en robotique. Le chapitre 3 décrit l'outils ROS-SNN. Les chapitres 4 et 5 présentent et analysent les résultats des tests effectués. Le chapitre 6 conclut le document.

CHAPITRE 2 LES RND ET LA ROBOTIQUE

La robotique et l'intelligence artificielle sont des domaines relativement jeunes. En 1950, Alan Mathison Turing posait une question qui allait jeter les bases de l'intelligence artificielle pour les décennies à venir : « *Can machines think?* » [1]. C'est également en 1950 qu'Isaac Asimov alimentait la réflexion sur la robotique. Le sujet était à la mode, mais relevait davantage de la science-fiction que de la science. Cependant, Turing a été le premier à tenter de définir sur quels critères l'humain pourrait évaluer l'intelligence d'une machine. C'est ainsi qu'il a proposé son célèbre Test de Turing. Depuis, plusieurs autres tests ont été proposés, tels que des tests d'employabilité, des examens scolaires et autres [11]. Les critères de Turing étaient en lien avec le fait que la machine puisse imiter l'humain dans sa finalité, mais pas nécessairement dans ses moyens pour y parvenir. Bref, il proposait un résultat qui imite l'intelligence humaine, sans pour autant y utiliser des moyens d'inspiration biologiques.

Les neurologues Warren McCulloch et Walter Pitts publièrent les premiers travaux sur des modèles mathématiques et logiques qui imitent le cerveau humain [4]. C'était le début des premiers réseaux de neurones artificiels. Cependant, cette première version de réseau de neurones était très limitée. Ils ne possédaient pas la capacité de s'adapter et de se modifier afin d'apprendre de nouvelles choses. En 1949, Donald Hebb (physiologiste canadien) avait proposé que la valeur des coefficients synaptiques se modifie au fil de l'apprentissage de l'humain [3]. Il jetait les bases à ce qu'allait devenir la plasticité. Ainsi est née la Règle de Hebb. Cette règle énonce que l'efficacité synaptique augmente lors d'une stimulation présynaptique répétée et persistante de la cellule postsynaptique. Elle suggère également que lorsque deux neurones sont excités ensemble, le lien entre eux va se créer ou va devenir plus fort s'il existe déjà. Frank Rosenblatt en 1957 s'inspira grandement de ce travail qui décrivait le cerveau humain pour l'appliquer aux réseaux de neurones artificiels. Il créa le perceptron [5], le premier système neuronal artificiel capable d'apprendre par ses expériences. Toutefois, ces réseaux avaient quelques limites [12]. Entre autres, un perceptron ne peut prendre que deux valeurs de sortie, soit 0 ou 1. Il ne peut aussi que faire un classement linéaire. Si un groupe de données ne peut être séparé de façon linéaire, l'apprentissage ne va jamais atteindre un point où le classement sera terminé.

Les faibles capacités de calcul et de traitement de l'information des ordinateurs de l'époque venaient restreindre grandement la recherche et les applications possibles. Les recherches sur les réseaux de neurones artificiels ont donc été sous financés pendant les années qui suivirent. Le domaine a connu un second souffle au début des années 80 avec les publications de John Joseph Hopfield et son Modèle de Hopfield [6]. Ce modèle a su regénérer de l'intérêt pour le domaine des réseaux de neurones artificiels. Il a innové en étant le premier modèle récurrent, avec les neurones connectés de façon non linéaire et où il y a au moins un cycle qui permet de boucler entre elles. Il y a aussi eu d'autres travaux intéressants, dont ceux de Geoffrey Hinton and Terry Sejnowski en 1985, qui ont créé la Machine de Boltzmann qui utilise une distribution mathématique de Boltzmann [7]. Il y eu ensuite une explosion de différents modèles, chacun appliqué à des applications spécifiques en traitement d'image, en traitement du signal, en classifications et en robotique. Le tout, avec des capacités de calcul toujours dérisoires, comparativement à la puissance de calcul du cerveau humain.

2.1 L'inspiration biologique des RND

Déjà, les réseaux de neurones artificiels étaient, jusqu'à un certain point, bio-inspirés. Les chercheurs ont poussé l'inspiration encore plus loin en créant des modèles qui utilisent également des décharges (des simulations d'impulsions électriques) ainsi qu'une dimension temporelle qui représentent encore mieux le fonctionnement du cerveau humain.

2.1.1 Le modèle Hodgkin-Huxley

Les travaux qui ont servi à créer les modèles artificiels de RND ont été fondés sur les travaux de Hodgkin et Huxley [2] [8] [13] [14]. Ces chercheurs ont été les premiers à analyser le fonctionnement de neurones sur un calmar géant. Ils ont décrit comment les potentiels d'actions sont déclenchés et transmis à travers le réseau. Ils ont également décrit comment les connexions synaptiques communiquent les neurotransmetteurs. Ce modèle va jusqu'à simuler les transferts d'ions de sodium et de potassium à travers les portes sur la membrane d'un neurone. L'équation (2.1) à la base du modèle de Hodgkin-Huxley montre bien la complexité de celui-ci. La variable I représente le courant total sur la membrane. C_m est la capacitance de la membrane alors que g_k et g_{Na} sont respectivement les conductances ioniques de potassium et de sodium. V_k et V_{Na} sont

les potentiels d'inversion de potassium et de sodium. Finalement, g_l est la conductance de fuite et V_l, le potentiel d'inversion de fuite. Le tout est exprimé en volt.

$$I = C_m \frac{\mathrm{d}V_m}{\mathrm{d}t} + g_K(V_m - V_K) + g_{Na}(V_m - V_{Na}) + g_l(V_m - V_l) \tag{2.1}$$

Encore aujourd'hui, il s'agit de l'un des modèles des plus complet quand il s'agit de reproduire le neurone biologique. Évidemment, ce modèle requiert une puissance de calcul faramineuse et ne suscite que peu d'intérêt pour la résolution de problèmes concrets en intelligence artificielle et en robotique. Les algorithmes sont tellement complexes qu'il faut des processeurs très puissants pour arriver à certains résultats. Ceci limite de facto leur exploitation en robotique due à l'impossibilité d'embarquer des processeurs puissants sur ces plateformes.

Somme toute, le modèle de Hodgkin et Huxley a été un modèle révolutionnaire pour bien comprendre le fonctionnement du cerveau. Il aura permis la création de modèles de RND plus simples qui eux sont plus appropriés pour une exploitation en robotique.

2.1.2 Le modèle *Leaky Integrate and Fire* (LI&F)

Le modèle de RND *Integrate and Fire* (I&F), dont le modèle biologique avait été créé initialement en 1907 par Louis Lapicque [15], a été repris ensuite par les chercheurs dont Izhikevich [9]. Le modèle I&F fait abstraction des détails de fonctionnement (porte ioniques, neurotransmetteurs, etc.) et ne simule que les potentiels d'action qui se propagent à travers le réseau. Les algorithmes qui implémentent ce modèle sont donc plus simples et en conséquence peuvent s'exécuter plus rapidement, ouvrant la possibilité d'être exploitable en robotique.

Dans la famille des modèles I&F, le modèle LI&F est sans doute le mieux connu. Un peu comme dans le cas d'un neurone biologique, le neurone d'un LI&F fuit, c'est-à-dire que son voltage décrémente dans le temps s'il n'est pas restimulé. La figure 2.1 démontre le fonctionnement du modèle [16]. La partie du haut de la figure illustre le fonctionnement du neurone biologique. La décharge (le *spike*) est transmise par l'axone d'un neurone, ensuite par la synapse vers les dendrites du neurone suivant dans le réseau. Les dendrites cumulent ensuite les signaux reçus vers le noyau du neurone. Les charges s'accumulent jusqu'à un certain seuil. Lorsque ce seuil est atteint, un potentiel d'action est à nouveau déclenché et envoyé sur l'axone. Le cycle continue. La partie du bas de la figure décrit le même processus que la partie du haut, mais à

l'aide d'un circuit électrique. L'équation (2.2) exprime les potentiels d'action dans le RND. Une résistance *R* est en parallèle avec une capacité *C*. L'intensité du courant *I* est la somme de l'intensité du courant qui passe par *R* et *C*, avec $I(t) = I_R(t) + I_C(t)$ selon la loi d'Ohm.

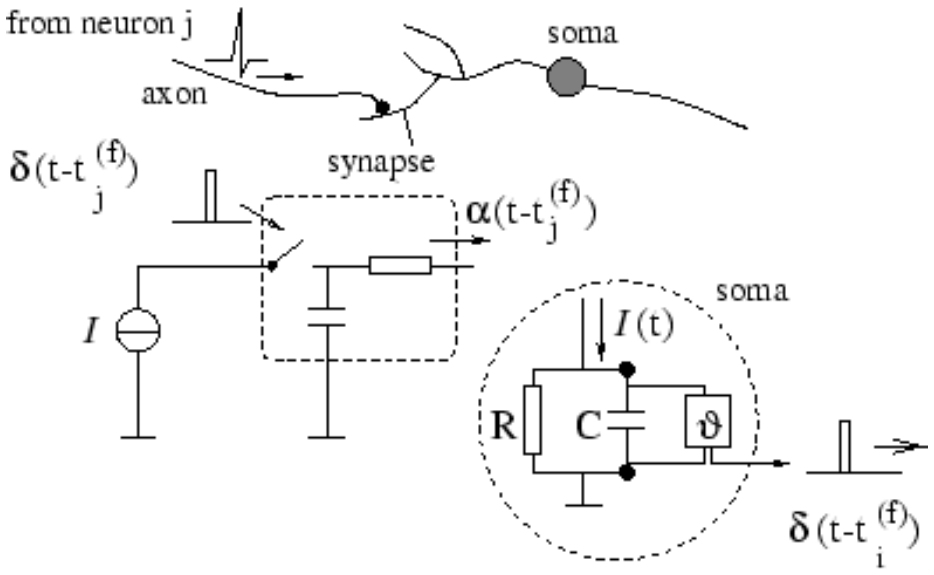


Figure 2.1: Le modèle LI&F [9]

Un graphe qui présente un RND compte trois types de neurones, définies en couches (*layers*) : les neurones sensoriels (*sensory*) en entrée, les neurones moteurs (*motor*) en sortie et les interneurones (*inter*). Tous les neurones situés entre les neurones sensoriels et les neurones moteurs sont des interneurones. Les neurones sensoriels reçoivent une certaine valeur exprimée en volt. Si la somme des valeurs reçues dépasse un certain seuil (*threshold*), un potentiel d'action (*spike*) est déclenché et se propage à travers les neurones liés et pondérés par les poids synaptiques. Les différents potentiels d'action, ou décharges, sont ainsi propagés à travers le RND jusqu'à atteindre les neurones moteurs. C'est d'ailleurs le contenu des neurones moteurs qui sont interprétés comme le résultat. Chaque neurone moteur qui déclenche ou non un potentiel d'action a une signification particulière (par exemple : avancer, arrêter, tourner un robot). L'équation différentielle qui détermine le voltage dans les neurones est donnée en (2.2).

$$\frac{\mathrm{d}v}{dt} = \frac{(I - v)}{tau} \tag{2.2}$$

La variable *I* (*Input drive current*) correspond à un certain courant qui arrive en continue aux neurones. La variable *v* représente le voltage, et la variable *tau* est un intervalle de temps. Après chaque potentiel d'action déclenché, le voltage du neurone sollicité est réinitialisé à une certaine valeur définie en paramètre. Celui-ci ne peut produire un nouveau potentiel d'action avant un certain laps de temps, aussi défini en paramètre.

2.1.3 Le modèle Izhikevich

Le modèle Izhikevich [9] a été créé pour tirer profit de la richesse des RND que peut générer le modèle Hodgkin-Huxley, tout en gardant un temps d'exécution rapide comme le fait le modèle LI&F. Ce modèle permet de simuler un RND en temps réel sur un simple ordinateur personnel.

Le modèle est simple et rapide à l'exécution. Tels les neurones biologiques, il peut produire de riches patrons de décharge. Un modèle I&F, en comparaison, est tout aussi rapide, mais ne peut produire des patrons aussi complexes et réalistes.

Les équations (2.3) à (2.4) présentent le modèle Izhikevich, qui simplifie plusieurs paramètres biophysique du modèle Hodgkin et Huxley :

$$v^1 = 0.04v^2 + 5v + 140 - u + I \tag{2.3}$$

$$u^1 = a(bv - u) \tag{2.4}$$

Finalement, l'équation (2.5) sert à la remise à zéro après décharge :

$$si\ v\ \geq 30\ mV, alors\ \begin{Bmatrix} v \leftarrow c \\ u \leftarrow u + d \end{Bmatrix} \tag{2.5}$$

Le paramètre a décrit l'échelle de temps de récupération de la variable u. Plus la valeur est petite, plus lente sera la récupération. Le paramètre b décrit la sensibilité de la variable de récupération u aux fluctuations du potentiel membranaire v. Le paramètre c est la valeur de réinitialisation du potentiel membranaire v (typiquement -65 mV). Le paramètre d détermine la valeur de réinitialisation après décharge de la variable u.

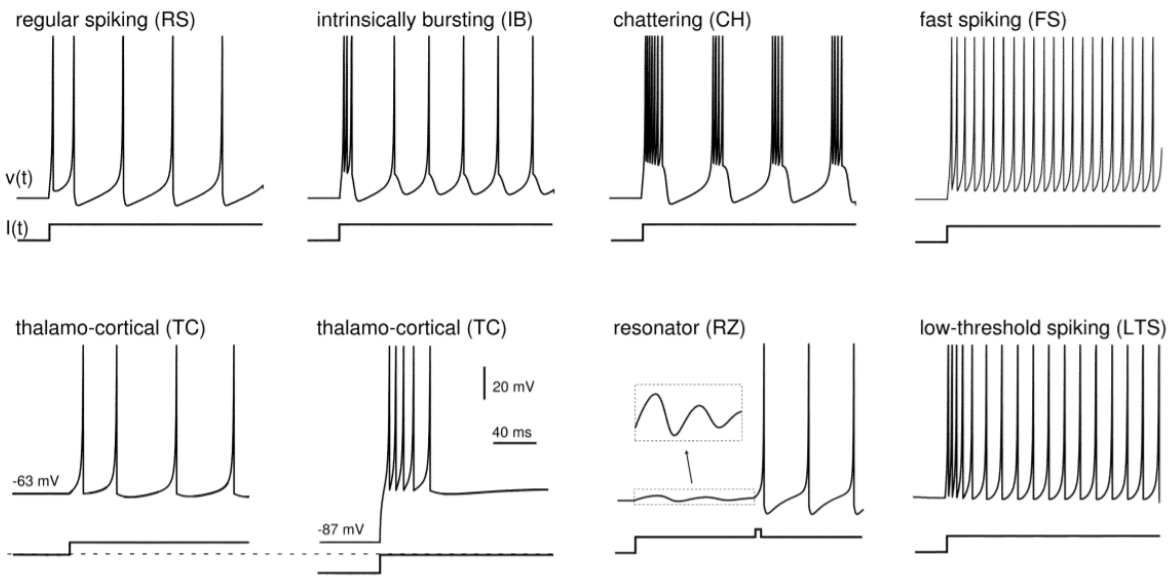


Figure 2.2: Types de neurones, selon ses paramètres [9]

Le modèle élabore une série de types de neurones définis en fonction de ses paramètres, comme le montre la figure 2.2. Ce modèle est très connu et a contribué de façon importante à l'évolution des RND.

2.2 Mise en œuvre des RND

A priori, les RND semblent présenter plusieurs désavantages : ils sont difficiles à implémenter, ils utilisent beaucoup de ressources à l'exécution (et en conséquence, ils sont plus lents), ils ne sont pas très connus. Alors pourquoi utiliser les RND? Parce que l'évolution en a fait le dénominateur commun des formes d'intelligence les plus complexes [17] [18]. Bien qu'il existe des formes d'intelligence simples qui ne sont pas basées sur les neurones [19], il n'existe pas de forme de raisonnement complexe et connu qui n'utilise pas de RND. La nature a pris des millions d'années pour façonner cette source d'intelligence, mais elle a réussi à y faire jaillir une intelligence générale.

Selon leur taille, les RND peuvent exiger une très grande capacité de calculs. Des recherches montrent que le traitement des RND en parallèle est efficace [20]. Pour ce faire, les grappes de serveurs sont souvent difficiles d'accès, spécialement lorsque le RND est embarqué sur une plateforme robotique. Une alternative intéressante aux grappes de serveur est sans doute l'utilisation des processeurs graphiques (GPU). Puisque ceux-ci sont conçus pour faire du

traitement de graphiques en parallèle, ils peuvent également exécuter d'autres tâches, tels que le calcul des équations de RND en parallèle. Ce support physique vient ainsi accélérer grandement le traitement.

Il existe des outils qui permettent de mieux intégrer les RND sur des GPU. Par exemple, NeMo [21] est une plateforme qui utilise les GPU pour le traitement des RND. Celui-ci implémente le modèle Izhikevich. Il peut créer un RND possédant 40 000 de neurones, avec 1 000 connexions synaptiques par neurone. La vitesse de déclenchement de décharge peut aller jusqu'à 10 Hz, ce qui signifie environ 400 millions de décharges par seconde. Le tout sur un support matériel qu'il est possible d'embarquer sur un robot.

Par contre, bien que les processeurs graphiques soient efficaces pour paralléliser le traitement, ils ne sont pas conçus spécifiquement pour traiter des RND. Des processeurs neuromorphiques, c'est-à-dire conçus avec une architecture qui implémente de façon matérielle les RND, commencent à apparaitre de façon expérimentale. Spécialement dans un contexte où la loi de Moore[2] tire à sa fin, il devient important de prévoir un support capable d'exécuter en temps réel un RND de grande taille. Pour reproduire les qualités du cerveau humain dans un processeur, il faut radicalement changer l'architecture de ce processeur. Contrairement à une machine d'architecture Von Neumann possédant un endroit pour traiter l'information et un endroit pour stocker l'information, le cerveau traite l'information là où elle est stockée. Ceci évite des milliards de transferts d'informations. Ce sont ces transferts d'information qui sont les plus énergivores dans une machine de type Von Neumann. L'absence de transfert d'information est la raison pour laquelle le cerveau humain consomme aussi peu d'énergie. Le défi est de reproduire les qualités du cerveau humain dans un processeur neuromorphique. Quelques travaux [22] [23] ont fait des percées significatives dans le domaine. Spécialement, le processeur neuromorphique *True North* d'IBM [24] est prometteur. Le SpiNNaker (*Spiking Neural Network Architecture)* [25] est un autre exemple d'architecture neuromorphique.

[2] Loi de Moore: Énonce que la capacité d'un ordinateur double à tous les deux ans. Il existe quelques précisions et quelques variantes de cette loi.

2.3 Utilisation des RND

Les RND sont utilisés dans plusieurs contextes, mais particulièrement ceux-ci :

- RND spécialisés : ils visent la reproduction de fonctionnalités avancées du cerveau à l'aide de modèles mathématiques. Les chercheurs créent de nouvelles équations aux modèles, les comparent, les analysent, les valident, etc. Ils évaluent les équations qui s'appliquent mieux à tel ou tel types de problèmes, ou qui s'appliquent le mieux à certaines parties du cerveau en particulier.
- RND généraux liés à la robotique : ils utilisent des équations simples afin d'appliquer les RND à la robotique. Certaines équations s'y prêtent mieux que d'autres. Plus une équation demeure simple, plus elle se prête à la robotique en fonction des capacités de calcul disponibles.

2.3.1 Les RND spécialisés

Un exemple d'un RND spécialisé [26] [27] est disponible sur le site de la librairie informatique Brian 2 [28]. Cette application modélise le système d'orientation d'un petit animal (un rongeur) à l'aide de ses moustaches et de la carte somatotopique de son réseau de neurones. Les moustaches du rongeur sont déplacées à l'aide de barres et l'impact de ces stimulations est observable sur la carte somatotopique. L'application tente d'implémenter les fonctionnalités du cortex cérébral en tenant compte des variables les plus importantes et en reproduisant le plus fidèlement possible les équations de génération de potentiel d'action. Elle tente également de reproduire le type de plasticité de cette partie spécifique du cerveau. Cette application tend à maximiser certains paramètres tels que les seuils adaptatifs de déclenchement des potentiels d'action, la durée de la période réfractaire du neurone, la valeur de réinitialisation après décharge, et autres.

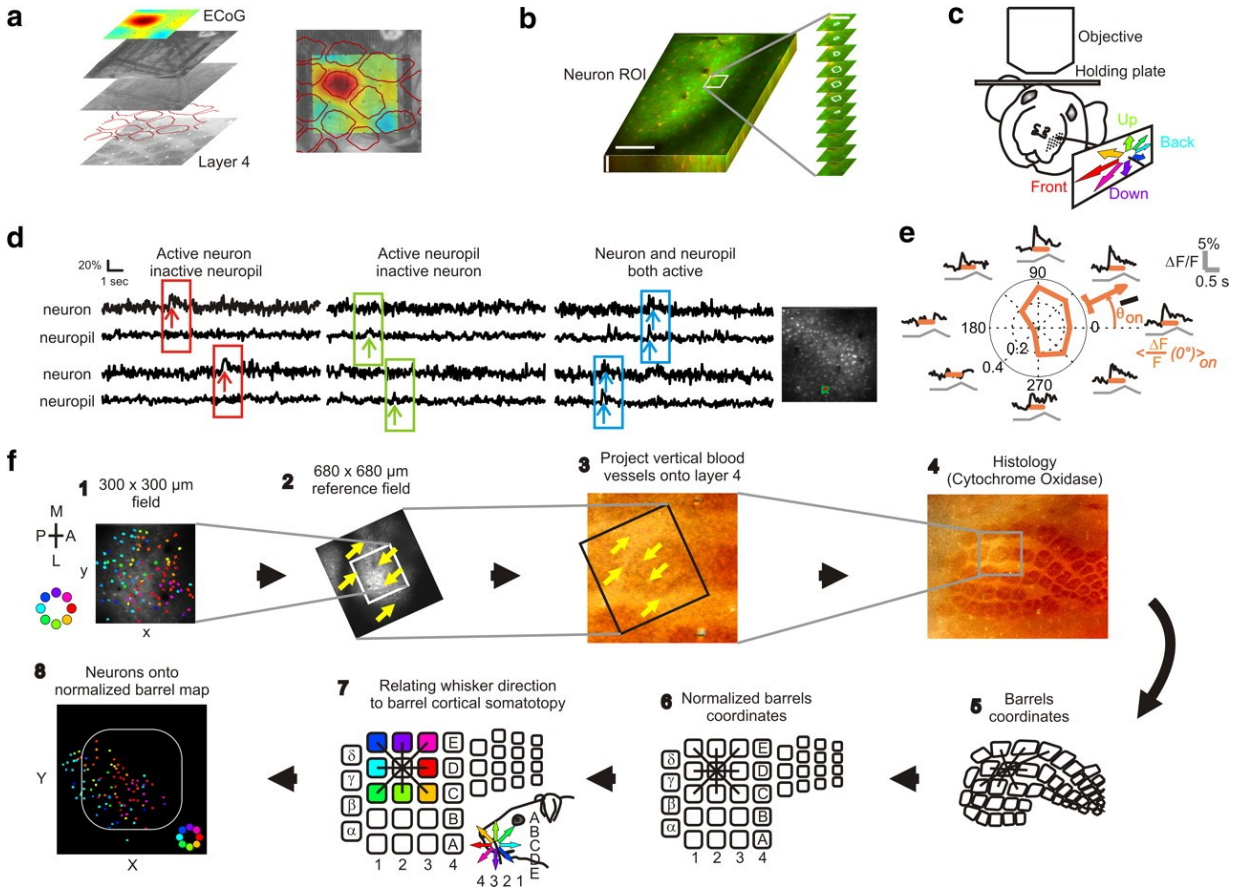


Figure 2.3: Modélisation des barils de neurones après la stimulation des moustaches d'un rongeur [27]

La figure 2.3 démontre le fonctionnement du RND de cette simulation :

a) À gauche, le montage montre un électrocorticogramme de 10 à 15 msec après que la moustache du rongeur ait dévié. Une série de couches modélise les différentes profondeurs du cortex. Les barils représentent les zones circulaires de stimulation propres à cette partie du cerveau, chez les rongeurs. La droite montre la superposition de ces images.
b) Image qui utilise la technique de microscopie d'excitation à deux photons. Celle-ci montre certaines parties d'un neurone lors de la stimulation des moustaches du rongeur. Il est à noter que la simulation Brian 2 ne produit que le RND de l'expérience.
c) Un code de couleurs pour illustrer la direction de déviation des moustaches.
d) Exemples d'activité neuronale et de ses neuropiles (fibres nerveuses) avoisinants.
e) Réponses directionnelles d'un neurone adulte typique.

f) Mise en commun de la cartographie à partir des différents champs et barils. Le point de départ (image f.1) est un plan cartésien des neurones. Le résultat final (image f.8) représente les neurones dans un plan cartésien normalisé et mettant en évidence la démarcation des barils.

Afin d'appuyer cette simulation, un exemple est implémenté à l'aide de la librairie Brian 2 [28]. Cet exemple utilise une RND de type I&F et une plasticité STDP. Toutefois, pour l'instant, il s'agit d'un type d'application qui n'a que peu d'intérêt pour la robotique. La puissance de calculs exigée est trop grande pour s'exécuter en temps réel sur une plateforme robotique. Cependant, au fur et à mesure que la puissance de calcul embarquée sur les robots va augmenter, une application comme celle-ci va devenir de plus en plus intéressante.

2.3.2 Les RND généraux et liés à la robotique

Les formules mathématiques derrières les RND appliquées à la robotique doivent demeurer relativement simples. Comme exemples de travaux particulièrement intéressants, Cyr et Boukadoum [29] [30] [31] [32] s'intéressent particulièrement aux RND appliqués à la robotique. Par exemple, l'un de ces projets s'intéresse au processus d'apprentissage des robots [32]. Dans la nature, l'apprentissage le fait grâce aux conséquences des actions précédentes ainsi qu'en corrélation passive d'évènement sans action. La prédiction des récompenses ou des punitions va motiver les comportements futurs. Dans un contexte de neurorobotique, les robots physiques et virtuels peuvent également apprendre de cette façon dans un environnement non supervisé. Cette étude démontre comment, avec différentes tâches d'un niveau de complexité variable, un RND de petite taille est suffisant pour réaliser plusieurs de ces apprentissages.

La figure 2.4 illustre les environnements d'expérimentation. Dans un environnement virtuel, un robot est composé d'une base cylindrique avec deux roues moteur, d'une roue avant et d'un bras sensoriel. Le robot physique est de type Lego Mindstorms NXT 2.0 et tend à reproduire le robot de l'environnement virtuel.

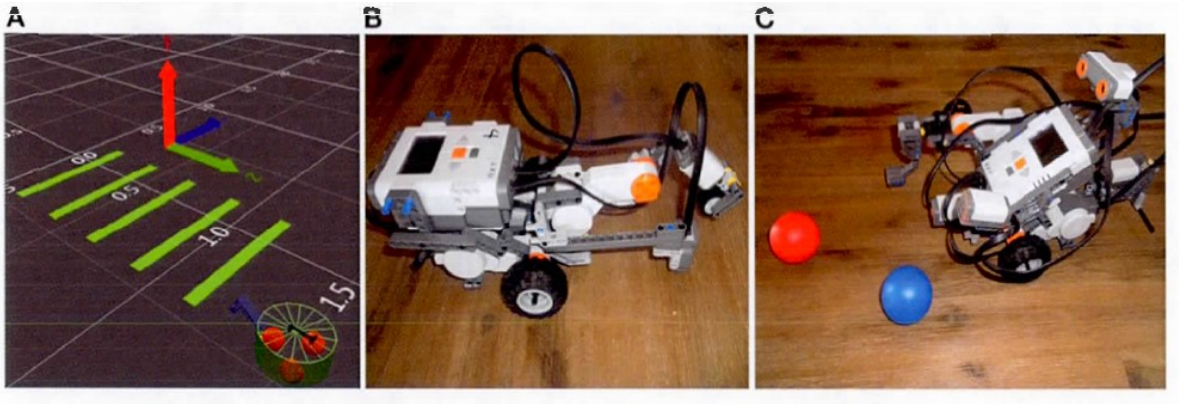


Figure 2.4 : Représentation virtuelle et physique des robots [32]

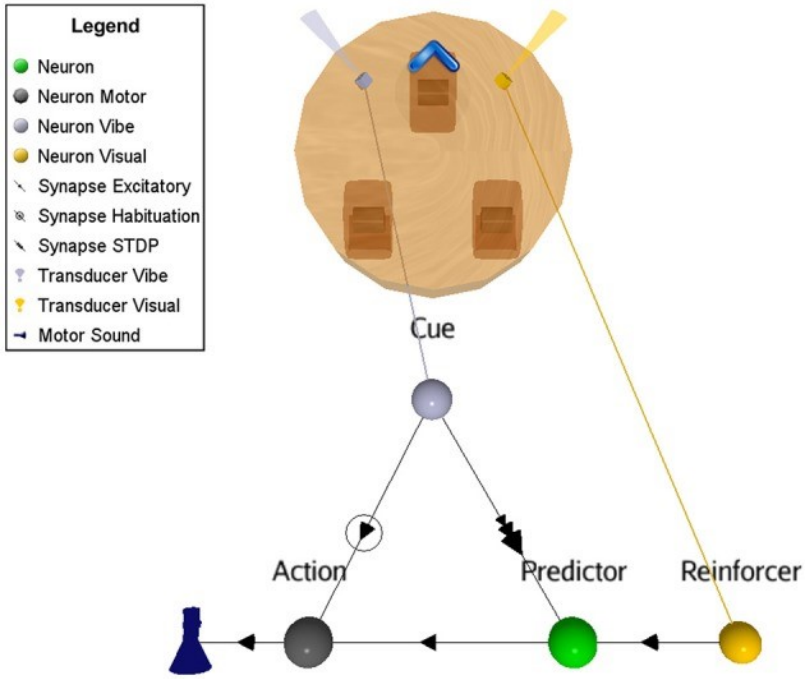


Figure 2.5 : Architecture du RND [32]

Dans sa version la plus simple, le RND compte seulement trois neurones, trois synapses, une règle d'habituation, et une règle de d'apprentissage synaptique STDP, comme le montre la figure 2.5. Le système n'utilise aucune équation différentielle, comme la plupart des RND. Il utilise des tables de référence (*lookup tables*) afin d'accélérer le traitement CPU, étant donné la faible puissance de calcul du robot.

L'étude réalisée propose quatre scénarios où le robot doit effectuer des tâches spécifiques. Les scénarios sont inspirés d'expériences souvent effectuées sur des animaux (spécialement sur les

rats et les pigeons). Le scénario A, par exemple, propose une expérience où le robot apprend par renforcement (positif ou négatif). Dans ce scénario, le renforcement positif se fait lorsque le neurone visuel (le jaune nommé *Reinforcer*) détecte de la lumière. Le neurone nommé *Cue* émet des décharges à intervalle constant. Une forte connexion synaptique entre les neurones *Cue* et *Action* commande au robot de faire les actions (produire un son, dans ce cas-ci). Cependant, cette connexion synaptique comprend une règle d'habituation qui fait faiblir la connexion dans le temps et éventuellement, les actions s'arrêtent. Pendant les 100 premiers cycles, le poids synaptique entre *Cue* et *Predictor* n'est pas assez fort pour produire des décharges. Graduellement pendant le processus d'apprentissage STDP, ce poids synaptique se renforci et ainsi le système peut générer des actions (des sons) à la détection de lumière. Ce RND bien entraîné peut continuer à commander des actions grâce au neurone nommé *Predictor*, et ce, même sans récompense.

CHAPITRE 3 LIBRAIRIE ROS-SNN

Les chercheurs qui veulent utiliser les RND sur des plateformes robotiques doivent constamment réinventer la roue. Il existe des librairies informatiques de RND sans toutefois qu'il y ait consensus sur laquelle utiliser ou qu'il y en ait une qui se démarche des autres. De plus, celles-ci ne sont pas adaptées du tout à des plateformes robotiques. Il existe également des librairies informatiques qui implémentent des fonctionnalités utiles à la robotique, mais ces librairies ne contiennent aucune fonctionnalité concernant les RND. Nous avons ainsi des outils pour l'un et pour l'autre, sans que celles-ci ne soient intégrées. Il n'existe aucun canal de communications entre ces outils robotiques et RND.

À cette fin, ce chapitre décrit la conception de la librairie ROS-SNN, un outil qui permet d'intégrer des RND sur des plateformes robotiques.

3.1 Spécifications

Le choix d'outils logiciels sur un robot mobile doit tenir compte de sa vitesse de traitement, et l'information doit souvent être traitée en temps réel. Par exemple, le robot doit naviguer en utilisant différents senseurs qui doivent être traités rapidement pour éviter les collisions. Il peut également avoir à répondre dans un court délai à une commande vocale. Idéalement, l'équipement embarqué sur un robot, incluant son unité centrale de traitement, doit être léger et consommer le minimum d'énergie possible lorsqu'il est alimenté par batteries. Si le robot doit avoir recours à une plus grande capacité de traitement, il peut avoir recours à un serveur par le biais d'un réseau. Cette solution a l'avantage de ne pas limiter la capacité de calcul embarquée sur le robot, mais il faut ajouter un certain temps de latence pour assurer le transfert des données par le réseau.

La librairie ROS [10] sert d'environnement de prototypage en robotique permettant de prendre en compte ces considérations. Distribué sous forme de logiciel ouvert, ROS permet de partager des routines de traitement et des fonctionnalités afin d'accélérer les développements en robotique. Cependant, ROS pourrait bénéficier de fonctionnalités associées à la mise en œuvre de RND. Il existe une multitude de paramètres à configurer pour utiliser efficacement les RND, connaissances que les roboticiens n'ont pas nécessairement pour utiliser convenablement les

RND. Arriver à implémenter une librairie de RND compatible avec ROS permettrait de rendre accessible des mises en œuvre de RND sur des plateformes robotiques. C'est l'objectif visé par la librairie ROS-SNN.

Les critères considérés pour la conception de la librairie ROS-SNN sont les suivants :

- Utiliser les langages Python ou C++, qui sont tous deux compatibles ROS et largement utilisés en robotique;
- Implémenter le modèle de RND LI&F, lequel est biologiquement réaliste et rapide. Les autres modèles tels celui de Hodgkin et Huxley auraient été tout aussi réalistes, mais demandent une capacité de traitement beaucoup plus grande puisque la simulation se fait jusqu'au niveau atomique;
- Être assez léger pour s'exécuter en temps réel sur un nano processeur de type ARMHF, comme par exemple le Raspberry Pi 3B;
- Être paramétrable, c'est-à-dire qu'avec des fichiers de définition de RND en format XML lancés avec les fichiers de lancement (*launchers*) de ROS, il faut que la librairie permette de définir une grande variété de configurations de RND;
- Permettre l'exécution en simultanée, en ayant plusieurs instances de la librairie exécutées simultanément avec des paramètres différents dans plusieurs nœuds ROS;
- Analyser le temps de traitement, soit examiner que le traitement du RND se fasse selon des contraintes temporelles. Sans offrir de mécanisme de contrôle temps réel, la librairie ROS-SNN doit examiner si le temps de traitement s'effectue en un temps moindre qu'un paramètre (en msec) défini par l'utilisateur.

Afin de ne pas avoir à réimplémenter sous ROS les fonctionnalités du modèle LI&F, une librairie déjà existante a été choisie. Avec les critères mentionnés plus haut, huit outils ont été évalués : Matlab [33], Simcog [34] , Genesis [35], SpiNNaker [25], Nest [36], CarlSim [37], Nemo [38] et Brian 2 [39]. Après évaluation, Brian 2 s'est révélé être le meilleur choix car en plus de répondre aux critères mentionnés, il est gratuit et distribué sous forme de logiciel ouvert, il est disponible sur la majorité des plateformes, il est léger en termes de calculs et implémente les fonctionnalités et concepts du modèle LI&F.

3.2 Fonctionnement du RND de la librairie ROS-SNN

La librairie ROS-SNN utilise le modèle LI&F, expliqué à la section 2.1.2. Voici comment la librairie ROS-SNN implémente les concepts de ce modèle.

D'abord, un graphe de RND est défini dans un fichier XML. Ce fichier XML contient l'architecture du RND (neurones et synapses disposés en couches) ainsi que chacune de ses propriétés. La puissance de la librairie ROS-SNN réside en sa capacité de créer différents RND sans même changer une ligne de code.

Un fichier de démarrage détermine les paramètres de lancement des deux nœuds nécessaires au traitement d'un RND. Le premier a pour but de déterminer quels sont les données à l'entrée qui alimentent les neurones sensoriels. Ce nœud normalise les données et les envoie sur un canal de communication ROS (*topic*). Le second nœud instancié par le fichier de démarrage représente le RND lui-même. C'est lui qui lit l'architecture et les propriétés du RND dans le fichier XML. Il instancie le RND et amorce une boucle de traitement infinie. À chaque itération, il lit les données reçues du premier nœud sur le canal de communication ROS. Il assigne ces données aux neurones sensoriels, et propage ces signaux dans le RND jusqu'aux neurones moteurs. Ces décharges neuronales sont ensuite envoyées sur les canaux de communication ROS de sortie, ou elles sont interprétées selon la signification qui leur est propre.

La librairie ROS-SNN offre également la possibilité de visualiser les flux d'information et l'architecture du RND à l'aide d'outils prévus à cet effet. Ces fonctionnalités sont expliquées aux sections 3.3 et 3.4.

3.3 Architecture de la librairie ROS-SNN

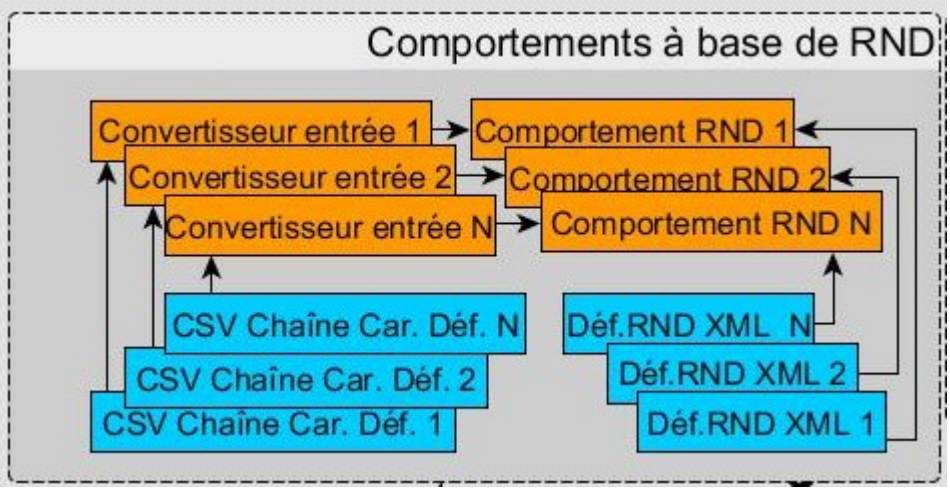


Figure 3.1: Architecture ROS-SNN

La figure 3.1 illustre l'architecture de la librairie ROS-SNN. Un fichier de lancement (*launcher* sous ROS) lance deux nœuds : un convertisseur d'entrée et un comportement RND. Ils peuvent être instanciés *N* fois, selon le nombre de RND à instancier.

- Un convertisseur d'entrée a pour but de lire les canaux de communication ROS (les *topics*) qui proviennent des senseurs, de convertir ainsi que de normaliser les signaux. Ces valeurs sont ensuite destinées à devenir les valeurs d'entrée des neurones sensoriels du RND. Ce premier nœud a pour but de publier les valeurs d'entrées pour le deuxième nœud (le RND), dans un format compatible. Si les valeurs reçues sont des chaines de caractères, les valeurs en nombre à point flottant sont lues à partir d'une table de correspondance dans un fichier CSV (désignée `CSV Chaîne de Car.Def`), grâce à des pairs « variables/valeurs ».
- Un comportement RND déclare l'architecture du RND dans le fichier `XML Déf.RND XML`. Le RND est généré dans une phase d'initialisation. Le RND démarre ensuite une boucle qui reçoit ses valeurs d'entrée normalisées (entre les valeurs des paramètres input_min_X et input_max_X) du premier nœud afin de stimuler son RND. Une séquence de potentiels d'action (*spikes*) est ensuite émise par les neurones moteurs, selon les valeurs d'entrées et les poids synaptiques spécifiés par l'architecture.

Plusieurs occurrences de RND peuvent s'exécuter simultanément.

La suite des explications se fait à partir d'un exemple fourni avec la librairie ROS-SNN et nommé `JoystickTest`. L'exemple vient simplement capter des signaux de la manette de jeu, converti ces signaux et les envoie à un RND[3]. Le but est d'illustrer de façon simple l'utilisation de l'outil ROS-SNN et toutes ses fonctionnalités. La sortie n'a que le sens que nous lui donnons, selon les paramètres d'entrée. Il s'agit d'une démo pour expérimenter les paramètres, et rien d'autre. De façon plus détaillée :

- Les axes (*thumbsticks*) gauche et droite de la manette envoient un signal sur les neurones sensoriels de 1 à 4.
    - Sur les axes X : Le signal est positif à droite et négatif à gauche.
    - Sur les axes Y : Le signal est positif en haut et négatif en bas.

La figure 3.2 représente le RND à implémenter, et la figure 3.3 l'architecture du comportement. La première partie du fichier de lancement ROS, donné à la figure 3.4, lance le nœud qui exécute le RND, soit le nœud `Comportement RND` de la figure 3.1.

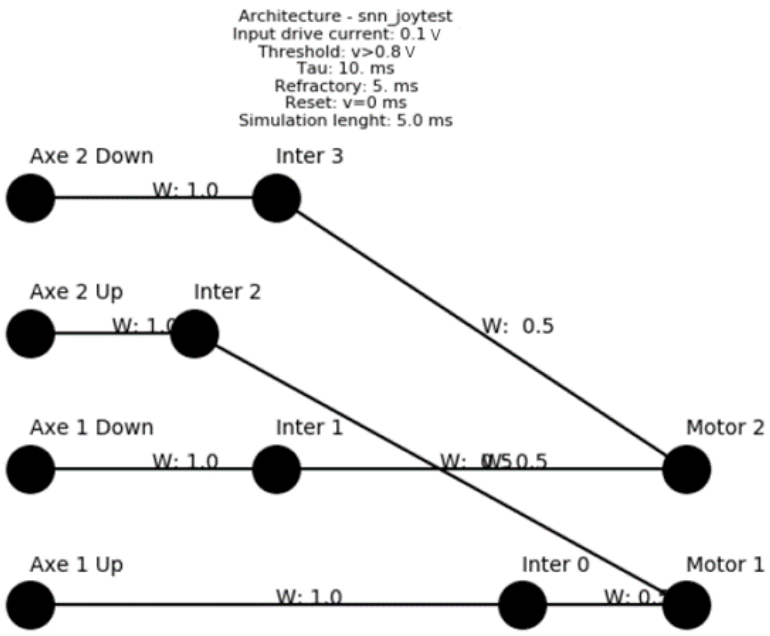


Figure 3.2: Architecture du RND `JoystickTest`

[33] Le code source est disponible ici : https://github.com/jsdessureault/ros_snn.

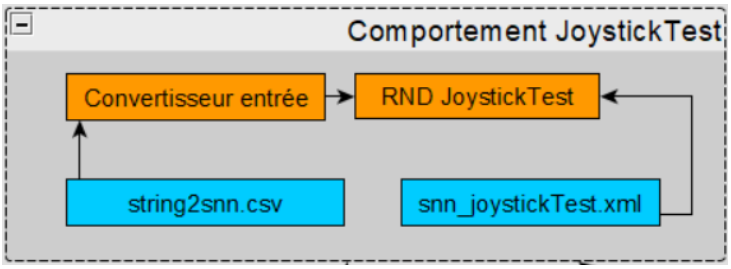


Figure 3.3: Architecture du comportement `JoystickTest`

```
<launch>
      <node name="SNN" pkg="ros_snn" type="SNN.py" output="screen"
      respawn="false">
            <!-- path where the XML can be found -->
            <param name="path"
            value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/>
             <!-- Name of the SNN -->
            <param name="SNNname" value="JoystickTest"/>
            <!-- Will display process if True -->
            <param name="verbose" type="bool" value="True"/>
            <!-- Name of the xml file -->
            <param name="xml" value="snn_JoystickTest.xml"/>
      </node>
```

Figure 3.4 : Première partie du code du fichier de lancement de `JoystickTest` (partie nœud `SNN`)

Les variables de nœud sont :

- `name`: Le nom du nœud. Doit toujours être égal à `SNN`.
- `pkg` : Le nom du package. Doit toujours être égal à `ros_snn`.
- `type` : Le nom du programme à exécuter. Doit toujours être égal à `SNN.py`.
- `output`: L'endroit de l'affichage des sorties standards. Doit toujours être égal à `screen`.
- `respawn`: Si la valeur `true` est spécifiée : dans le cas où le nœud s'arrête, celui-ci va redémarrer automatiquement.

Les paramètres à spécifier sont:

- `Path`: Le chemin du fichier XML, en excluant `/xml`. Si le fichier est `/home/pi/ros_catkin_ws/src/ros_snn/snn/xml/snn_JoystickTest.xml`, la variable `Path` aura comme valeur `/home/pi/ros_catkin_ws/src/ros_snn/snn/`.

- `SNNname`: Le nom du RND. Ce nom doit être unique si plusieurs RND s'exécutent de façon simultanée.
- `verbose`: Indique si le nœud va afficher des messages d'information à la console.
- `xml`: Le nom du fichier XML qui définit l'architecture du RND. La totalité du RND y est défini dans ce seul et même fichier.

La deuxième partie du fichier de lancement ROS est présentée par la figure 3.5. Elle démarre le nœud de conversion des données provenant des senseurs. Elle convertie et normalise l'information reçue sous forme de *topics* en données à utiliser directement par les neurones sensoriels du nœud `Comportement RND`.

Les variables de nœud sont :

- `name`: Le nom du nœud. Doit toujours être égal à `input_converter`.
- `pkg` : Le nom du package. Doit toujours être égal à `ros_snn`.
- `type` : Le nom du programme à exécuter. Doit toujours être égal à `input_converter.py`.
- `output`: L'endroit de l'affichage des sorties standards. Doit toujours être égal à `screen`.
- `respawn`: Si la valeur `true` est spécifiée : dans le cas où le nœud s'arrête, celui-ci va redémarrer automatiquement.

Les paramètres du nœud sont :

- `SNNname`: Le nom du RND. Ce nom doit être unique si plusieurs RND s'exécutent de façon simultanée.
- `verbose`: Indique si le nœud va afficher des messages d'information à la console.
- `topics_to_convert`: Le nombre de topics à convertir. Pour chaque topic à convertir, il faut spécifier un bloc de paramètres (X étant le numéro du paramètre):
    - `input_topic_X`: Le nom du *topic* à lire en entrée.
    - `topic_type_X`: Le type du *topic* à lire en entrée (p.ex. : Joy, Float32, Int16…)
    - `Input_field_X`: Le champ du *topic* qui contient la donnée (p.ex.: axes[4], .data…)

- `input_min_X`: La valeur minimum possible, afin de normaliser l'entrée.
- `input_max_X`: La valeur maximum possible, afin de normaliser l'entrée.

Outre le contenu des neurones moteurs, il est possible de vérifier le contenu du topic ROS nommé `realtime_<<nom_du_test>>`. Celui-ci a une valeur négative si le temps d'exécution a lieu en deçà de la valeur prévue par le paramètre `Realtime_limit`. Si la valeur excède ce paramètre, le *topic* ROS retourne une valeur positive, qui indique que le cycle dépasse sa cible de X msec.

Considérons que la manette de jeu diffuse un *topic* ROS appelé `/joy` avec un de ses champs nommé `axes`, référé par `/joy.axes[x]`. Ces champs d'axes contiennent la valeur des différents axes de la manette de jeu. Le domaine est un nombre réel sur une plage de -1.0 à 1.0. Leur valeur est normalisée entre `input_min` (-1.0) et `input_max` (1.0) afin d'être utilisée par le RND.

Ce nœud de conversion d'entrée crée des nouveaux *topics* en tenant compte du nom du RND, selon le format *(SNNname)_(input_number)_snn_in* (p.ex. : `JoystickTest_1_snn_in`, `JoystickTest_2_snn_in`, `JoystickTest_3_snn_in` et `JoystickTest_4_snn_in`).

```
<node name="input_converter" pkg="ros_snn" type="input_converter.py"
output="screen"respawn="false">
      <!-- Name of the SNN -->
      <param name="SNNname" value="JoystickTest"/>
      <!-- Will display process if True -->
      <param name="verbose" type="bool" value="True"/>
      <!-- Number of sensory neurons -->
      <param name="topics_to_convert" type="int" value="4"/>

      <!—For each of the following parameters:  -->
      <!—input_topic_X: Topic containing the value of the input neuron -->
      <!—topic_type_X: Topic containing the type of the input neuron topic
      1-->
      <!—Input_field_X: Field of the topic containing the value of the input
      neuron -->
      <!—input_min_X: Minimum value of the MinMax function -->
      <!—input_max_X: Minimum value of the MinMax function -->

      <param name="input_topic_1" value="/joy"/>
      <param name="topic_type_1" value="Joy"/>
      <param name="input_field_1" value=".axes[1]"/>
      <param name="input_min_1" value="-1.0"/>
      <param name="input_max_1" value="1.0"/>

      <param name="input_topic_2" value="/joy"/>
      <param name="topic_type_2" value="Joy"/>
      <param name="input_field_2" value=".axes[0]"/>
      <param name="input_min_2" value="-1.0"/>
      <param name="input_max_2" value="1.0"/>

      <param name="input_topic_3" value="/joy"/>
      <param name="topic_type_3" value="Joy"/>
      <param name="input_field_3" value=".axes[4]"/>
      <param name="input_min_3" value="-1.0"/>
      <param name="input_max_3" value="1.0"/>

      <param name="input_topic_4" value="/joy"/>
      <param name="topic_type_4" value="Joy"/>
      <param name="input_field_4" value=".axes[3]"/>
      <param name="input_min_4" value="-1.0"/>
      <param name="input_max_4" value="1.0"/>
</node>
</launch>
```

Figure 3.5 : Deuxième partie du code du fichier de lancement de `JoystickTest` (partie nœud `input_converter`)

Si des chaines de caractères sont les entrées du RND, le niveau de voltage équivalent doit être lu dans un fichier .CSV qui contient des pairs variables/valeurs, comme dans l'exemple suivant, défini à l'avant-dernière ligne. Le dernier paramètre est `stringfile_1` et sa valeur est `string2snn.csv`. Il s'agit d'un fichier avec pairs variables/valeurs, séparées par des virgules. Le nom de la variable représente la chaîne de caractères, et la valeur représente le voltage associé

à cette variable. Il s'agit du lancement du nœud `Convertisseur entrée` et du fichier CSV `Chaîne car. Déf` de la figure 3.1. La figure 3.6 présente la déclaration de ce nœud dans un fichier de démarrage.

```
<node name="input_converter" pkg="ros_snn" type="input_converter.py"
output="screen" respawn="false" required="true">
      <!-- Name of the SNN -->
      <param name="SNNname" value="stringtest"/>
      <!-- Will display process if True -->
      <param name="verbose" type="bool" value="True"/>
      <!-- Number of sensory neurons -->
      <param name="topics_to_convert" type="int" value="1"/>
      <!-- path where the trained SNN is saved -->
      <param name="path" value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/>

      <!-- Topic containing the value of the input neuron -->
      <param name="input_topic_1" value="/strcommand"/>
      <!-- Topic containing the type of the input neuron topic 1-->
      <param name="topic_type_1" value="String"/>
      <!-- Field of the topic containing the value of the input neuron -->
      <param name="input_field_1" value=".data"/>
      <!-- Definition of the strings (if defined) -->
      <param name="stringfile_1" value="string2snn.csv"/>
</node>
```

Figure 3.6 : Deuxième partie du code du fichier de lancement de `JoystickTest` (partie nœud `input_converter`) avec convertisseur de chaînes de caractères

Dans l'exemple précédent, notez qu'une variable nommée `stringfile_1` contient un fichier nommé `string2snn.csv`. Ce fichier de pairs `variables/valeurs` pourrait ressembler à celui de la figure 3.7. Ceci signifie que, si par exemple, un *topic* nommé `/strcommand` reçoit la chaîne de caractères `Green` dans son champ nommé `.data`, il enverra une valeur de 0.5 au RND.

```
string,voltage
Red,0.1
Green,0.5
Blue,1.0
```

Figure 3.7 : Code du fichier `string2snn.csv`

La commande pour démarrer le fichier de lancement `JoystickTest.launch` dans ROS est la suivante :

```
roslaunch ros_snn behavior_JoystickTest.launch
```

La commande pour démarrer le fichier de lancement `stringtest.launch` dans ROS est la suivante :

```
roslaunch ros_snn behavior_stringtest.launch
```

L'architecture du RND doit être déclarée dans un fichier XML en utilisant des balises spécialement prévues à cet effet. Chaque neurone, chaque synapse, chaque couche et chaque paramètre doivent être déclarés dans ce fichier. Au besoin, une même architecture peut être instanciée plusieurs fois en ayant des noms et des paramètres différents. La figure 3.8 présente la déclaration typique d'un RND dans un fichier XML, selon l'architecture de la figure 3.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<SNN name="snn_JoystickTest" realtime_limit="0.75"
synapse_delay="1.0" input_drive_current="0.1" tau="10"
threshold="v>0.8" reset="v=0" refractory="5" sim_lenght="50">

     <layer type="sensory" name="sensory">
          <neuron id="0">Axis 1 Up</neuron>
          <neuron id="1">Axis 1 Down</neuron>
          <neuron id="2">Axis 2 Up</neuron>
          <neuron id="3">Axis 2 Down</neuron>
     </layer>

     <layer type="inter" name="inter">
          <neuron id="0" synapse="0" layer="sensory"
weight="1.0">Inter
0</neuron>
          <neuron id="1" synapse="1" layer="sensory"
weight="1.0">Inter 1</neuron>
          <neuron id="2" synapse="2" layer="sensory"
weight="0.2">Inter 2</neuron>
          <neuron id="3" synapse="3" layer="sensory"
weight="0.2">Inter 3</neuron>
     </layer>

     <layer type="motor" name="motor">
          <neuron id="0" synapse="0, 1" layer="inter" weight="0.5,
0.5">Axis 1</neuron>
          <neuron id="1" synapse="2, 3" layer="inter" weight="0.5,
0.5">Axis 2</neuron>
     </layer>
</SNN>
```

Figure 3.8: Code du fichier `snn_JoystickTest.xml`

Un fichier XML est composé de balises, qui permettent de modéliser un concept. Dans le cas de la librairie ROS_SNN, le concept à définir est un RND. Le graphe de RND est défini à l'aide

des balises SNN, LAYER et NEURON. Chaque balise possède des paramètres qui lui sont propres. Les tableaux 3.1, 3.2 et 3.3 expliquent les paramètres du RND dans le fichier XML.

Tableau 3.1: Paramètres de la balise SNN tels que définis à la figure 3.8

Paramètres	**Description**
`name`	Le nom unique du RND.
`realtime_limit`	Une limite supérieure, exprimé en msec, qui représente le temps maximum acceptable d'un cycle. Le concepteur du RND pourra vérifier la sortie du RND dans un *topic* nommé `realtime_(nom_du_RND)`. La valeur de sortie représente le temps d'exécution du dernier cycle moins le temps de cycle limite. Si la valeur produite est négative, la valeur limite est respectée. Si la valeur produite est positive, le temps de cycle excède le temps limite. La valeur visée est la plus basse possible, dans les nombres négatifs. Par exemple : • -30 : le temps d'exécution réel est de 30 msec de moins que le temps limite spécifié. • 5 : le temps d'exécution réel est de 5 msec de plus que le temps limite spécifié.
`synapse_delay`	Un court délai exprimé en msec. Sans ce délai, les potentiels d'action des différents neurones sont parfaitement synchronisés, ce qui rend leur affichage plus difficile sur les graphiques. Les courbes de voltage se superposent. Avec ce court délai entre chacun, ceux-ci ne se superposent pas sur les graphiques.
`input_drive_current`	Un voltage constant ajouté aux neurones d'entrées afin de mieux simuler les neurones biologiques. Il arrive que des neurones reçoivent un stimuli constant.

`tau`	Une valeur en msec utilisée dans l'équation. Cette valeur a pour but de garder la consistance entre les deux parties de l'équation.
`threshold`	Le seuil de voltage auquel les neurones engendrent un potentiel d'action.
`reset`	La valeur de réinitialisation à la suite du potentiel d'action.
`refractory`	Une période de repos en msec suite à un potentiel d'action. Pendant cette période, aucun autre potentiel d'action ne pourra être déclenché.
`sim_lenght`	La durée de la simulation en msec.

Tableau 3.2: Paramètres de la balise LAYER

Paramètres	**Description**
`type`	Doit être `sensory`, `inter` ou `motor`.
`name`	Un nom unique pour chacun.

Tableau 3.3: Paramètres de la balise NEURON

Paramètres	**Description**
`id`	Un identificateur de neurone unique à l'intérieur d'une couche.
`synapse`	Une synapse qui se connecte à un autre neurone d'une autre couche. Ce peut également être plusieurs synapses en spécifiant une liste (p.ex. `1, 2, 3`).
`layer`	Les couches où se connectent les synapses.
`weight`	Le poids de la synapse. Dans le cas de plusieurs synapses, une liste de poids doit être spécifiée (p. ex. `0.2, 0.6, 0.4`).
`<Texte libre dans la balise>`	Le nom du neurone, comme `Axis 1`.

Par exemple, la définition suivante indique que le neurone se nomme `Axis 1`, avec un identificateur unique de 0, ayant des synapses qui réfèrent aux neurones 0 et 1 de la couche `inter` et ayant des poids synaptiques de 0,5 et 0,5.

```
<neuron id="0" synapse="0, 1" layer="inter" weight="0.5, 0.5"> Axis
1</neuron>
```

3.4 Outil de visualisation des RND

La librairie ROS-SNN est conçue pour offrir un outil de visualisation des RND. Il permet au développeur d'analyser l'architecture du RND, les signaux d'entrée, ainsi que les potentiels d'action des neurones moteurs du RND. Un fichier de lancement (*launcher*) ROS doit être appelé en spécifiant trois paramètres pour chaque nœud à l'intérieur de celui-ci, comme dans le code de la figure 3.9 :

```
<launch>
 <node name="plot_volts" pkg="ros_snn" type="plot_volts.py" output="screen"
 respawn="false">
  <!-- path where the trained SNN is saved -->
  <param name="path" value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/>
  <!-- Name of the xml file -->
  <param name="xml" value="snn_JoystickTest.xml"/>
  <!-- Name of the SNN -->
  <param name="SNNname" value="JoystickTest"/>
 </node>
 <node name="plot_spikes" pkg="ros_snn" type="plot_spikes.py"
output="screen" respawn="false">
  <!-- Name of the SNN -->
  <param name="SNNname" value="JoystickTest"/>
  <!-- path where the trained SNN is saved -->
  <param name="path" value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/>
  <!-- Name of the xml file -->
  <param name="xml" value="snn_JoystickTest.xml"/>
 </node>
 <node name="plot_input" pkg="ros_snn" type="plot_input.py" output="screen"
respawn="false">
  <!-- path where the trained SNN is saved -->
  <param name="path" value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/>
  <!-- Name of the xml file -->
  <param name="xml" value="snn_JoystickTest.xml"/>

  <!-- Name of the SNN -->
  <param name="SNNname" value="JoystickTest"/>
 </node>
 <node name="plot_connectivity" pkg="ros_snn" type="plot_connectivity.py"
output="screen" respawn="false" required="true">
  <!-- path where the trained SNN is saved -->
  <param name="path" value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/>
  <!-- Name of the SNN -->
  <param name="SNNname" value="JoystickTest"/>
  <!-- Name of the xml file -->
  <param name="xml" value="snn_JoystickTest.xml"/>
 </node>
</launch>
```

Figure 3.9 : Fichier de lancement de l'outil de visualisation.

Dans ce fichier de démarrage, quatre nœuds sont démarrés. Chaque nœud affiche un graphique sur les différents aspects du RND. Les variables de nœud sont :

- `name`: Le nom du nœud. Les quatre nœuds démarrés sont `plot_volts`, `plot_spikes`, `plot_input` et `plot_connectivity`.
- `pkg` : Le nom du package. Doit toujours être égal à `ros_snn`.
- `type` : Le nom du programmer à exécuter. Doit toujours être égal à `plot_volts.py`, `plot_spikes.py`, `plot_input.py` et `plot_connectivity.py`.

- `output`: L'endroit de l'affichage des sorties standards. Doit toujours être égal à `screen`.
- `respawn`: Si la valeur `true` est spécifiée : dans le cas où le nœud s'arrête, celui-ci va redémarrer automatiquement.

Le tableau 3.4 présente la description des paramètres :

Tableau 3.4: Paramètres du lancement des outils de visualisation

Paramètres	**Description**
`path`	Le chemin où se trouve le RND dans l'espace de stockage.
`SNNName`	Le nom unique du RND
`xml`	Le nom du fichier XML où se trouve la définition du RND.

Le lancement de l'outil se fait à l'aide de la ligne suivante :

```
roslaunch ros_snn (your_launch_file).launch
```

Les figures 3.2, 3.10, 3.11 et 3.12 illustrent des exemples de graphiques générés par l'outil de visualisation de l'outil ROS-SNN. La figure 3.2 est issue du type `plot_connectivity` et permet de visualiser le l'architecture du RND. Le fichier de définition `.xml` est lu et affiché sous forme de graphique. Celui-ci indique les neurones sensoriels, les interneurones, les neurones moteurs et les poids synaptiques entre chacune. Les paramètres du RND sont également affichés en entête de graphique. Le nom du RND est indiqué, le niveau de courant continue (*input drive current*), le seuil de déclenchement d'une décharge (*threshold*), la variable *tau* qui indique une unité de temps, une période de réfraction (*refractory period*) en deçà duquel aucun nouveau potentiel d'action n'est possible, une valeur de remise à zéro (*reset*) et la longueur d'une simulation (*simulation length*).

Montré à la figure 3.10, le graphique de type `plot_input` permet de visualiser le voltage reçu en entrée, par neurone. Les valeurs sont directement saisies à partir du *topic* sur lequel la valeur est émise par le nœud de conversion. Les valeurs sont normalisées (entre 0 et 1), et le mouvement haut/bas et gauche/droite des axes de la manette est visible.

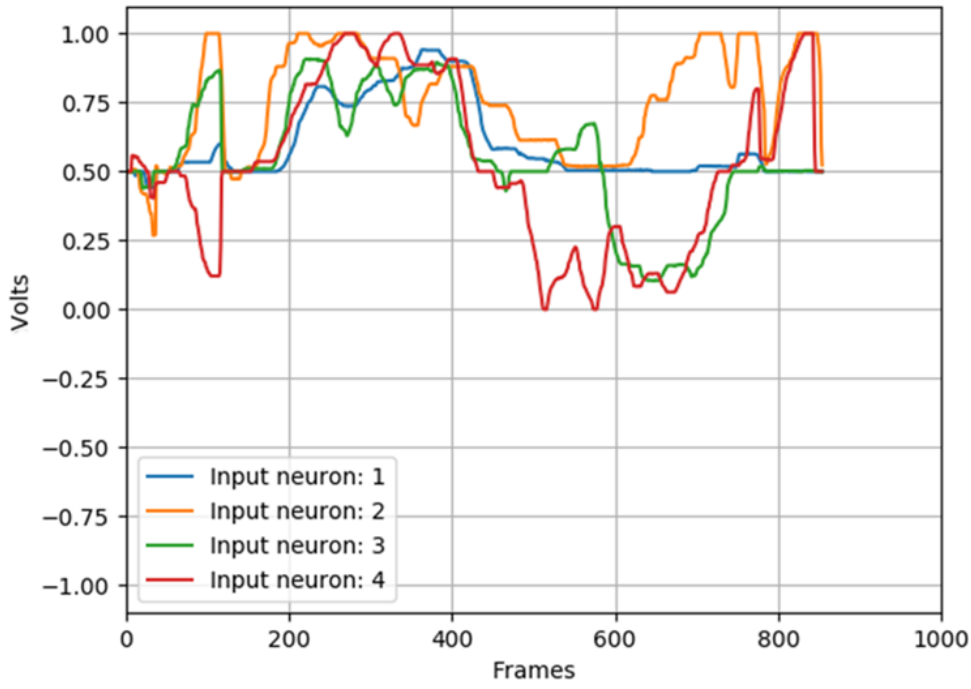


Figure 3.10: Voltage des neurones sensoriels

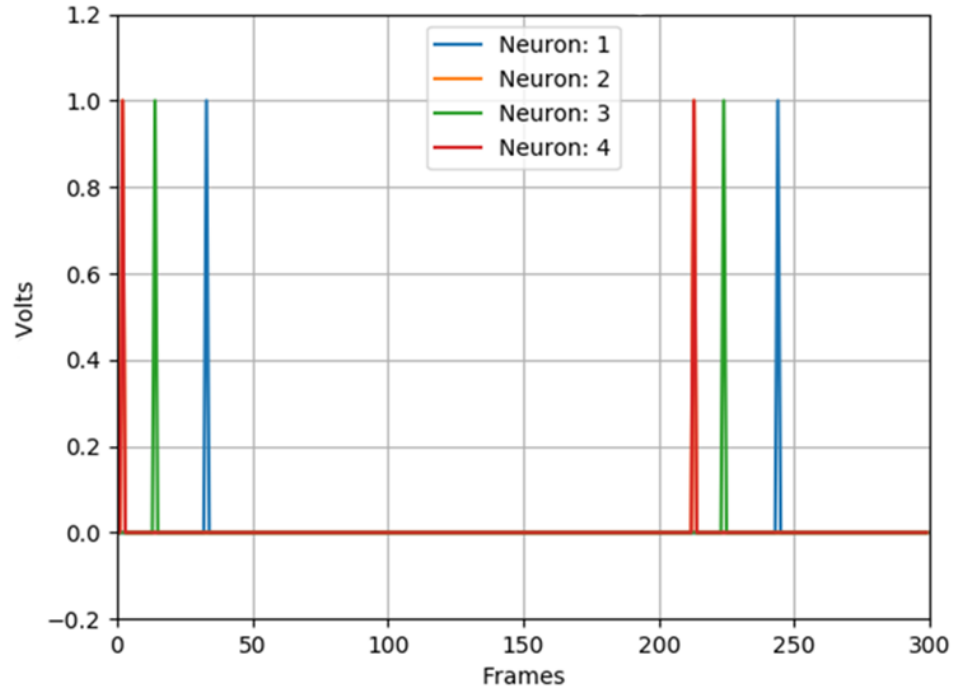


Figure 3.11: Voltage des neurones moteurs

La figure 3.11 montre le graphique de type `plot_volt`, permettant de visualiser le voltage calculé dans les neurones moteurs, à la sortie. Les valeurs sont directement émises dans les

topics moteurs. Les neurones moteurs de 1 à 4 déchargent à des moments différents. Les décharges (ou potentiel d'action) sont représentées par des sommets (*spikes*) de voltage dans le graphique. Ces valeurs émises par les neurones moteurs sont celles qui seront interprétées à la sortie du RND. Une petite valeur spécifiée par la variable « synapse_delay » fait en sorte que les décharges de neurones ne vont pas s'afficher au même endroit sur le graphique. Un neurone moteur qui décharge pourrait être interprété comme une commande de virage à gauche, par exemple.

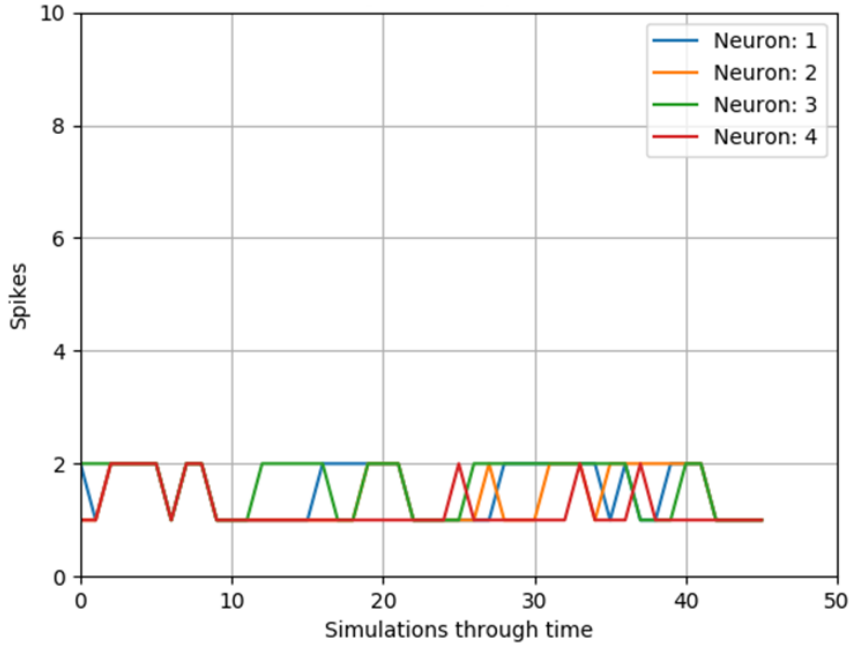


Figure 3.12: Décharges des neurones moteurs

La figure 3.12 montre le graphique de type `plot_spike` utilisé pour visualiser le nombre de décharges par neurone moteur, à la sortie. Les valeurs sont directement émises dans les *topics* moteurs. Les neurones de 1 à 4 déchargent à des moments différents. Les décharges (ou potentiel d'action) sont représentées par des sommets (*spikes*) de voltage dans le graphique. Contrairement à la figure 3.11, les valeurs sont exprimées en nombre de décharges, et non en voltage. Ces valeurs émises par les neurones moteurs sont celles qui seront interprétées à la sortie du RND. Un neurone moteur qui décharge pourrait être interprété comme une commande de virage à gauche, par exemple.

Tout ces graphiques sont particulièrement utiles pour la phase de déverminage d'un RND.

CHAPITRE 4 IMPLÉMENTATION D'UN RND SUR UN ROBOT MOBILE AVEC ROS-SNN

Afin de démontrer toutes les fonctionnalités de l'outil ROS-SNN, trois tests sont effectués sur le robot Spike. Le premier test utilise l'exemple `JoystickTest` traité au chapitre 3 et inclut dans le package ROS-SNN. Il capte les signaux d'entrée sur la manette de jeu et les envoie au RND. Le deuxième test gère la navigation du robot Spike.

4.1 Test d'un RND qui reçoit des commandes de la manette de jeu

Le RND `JoystickTest` capte simplement des signaux de la manette de jeu, converti ces signaux et les envoie à un RND[4]. L'intérêt de ce test est de dégager un ordre de grandeur des temps de traitement nécessaire à l'exécution des cycles d'une RND sur un nano processeur. Utilisant le RND de la figure 3.2, nous avons fait une étude examinant les charges en termes de temps de cycle de calcul en fonction des différents paramètres d'un RND, soit le nombre de neurones sensoriels, le nombre d'interneurones, le nombre de couches, et le nombre de neurones moteurs, pris individuellement, ainsi que le temps de cycle et le nombre de RND exécuté en parallèle. Les temps *Min* et *Max* sont examinées : *Min* représente le temps minimal d'un cycle, lorsque les senseurs n'envoient aucune valeur au RND; *Max* représente le temps d'un cycle lorsque les senseurs envoient un maximum de données au RND. Le temps de cycle varie entre ces valeurs *Min* et *Max*, selon la concentration de données provenant des senseurs. Les tests ont été faits sur un Raspberry Pi 3B.

[4] Le code source est disponible ici : https://github.com/jsdessureault/ros_snn.

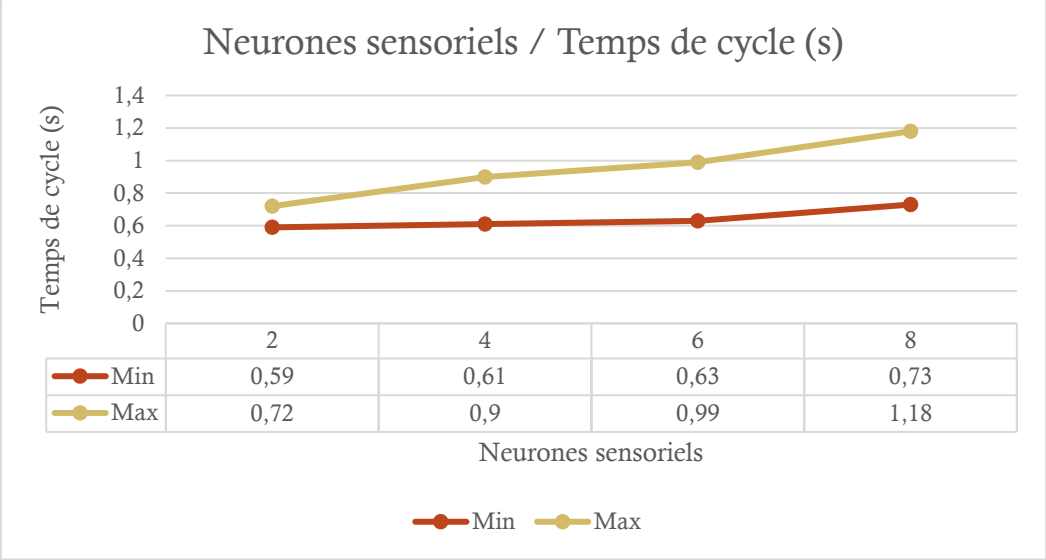


Figure 4.1: Impact du nombre de neurones sensoriels sur le temps de cycle

La figure 4.1 illustre le temps de cycle en modifiant le nombre d'entrées dans le RND de la figure 3.2. Le temps de cycle augmente de façon linéaire avec le nombre de neurones d'entrées, mais les droites *Min* et *Max* tendent à diverger avec l'augmentation du nombre de neurones. La raison est que chaque neurone d'entrée est associé à un *topic* sous ROS, ce qui fait augmenter le coût de son utilisation, spécialement quand le *topic* est utilisé à son maximum.

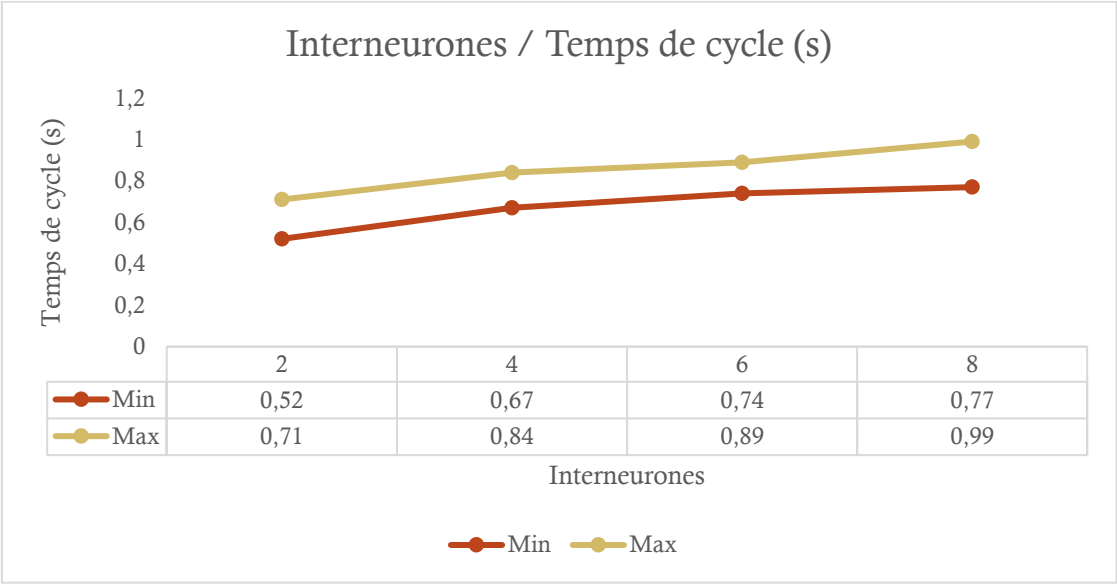


Figure 4.2 : Impact du nombre d'interneurones sur le temps de cycle

La figure 4.2 montre l'impact du nombre d'interneurones sur le temps de cycle. Le nombre d'interneurones ajoute de la flexibilité au RND : plus il y a d'interneurones, plus le signal est pondéré. Tout comme au sujet des neurones sensoriels, il y a un certain coût à ajouter des interneurones. L'augmentation est aussi linéaire, mais contrairement au nombre de neurones sensoriels, les droites *Min* et *Max* restent parallèles. La raison est qu'il n'y a pas plus ou moins de *topics* utilisés lors de l'usage des interneurones. Les interneurones sont totalement indépendants des *topics* et de leur utilisation.

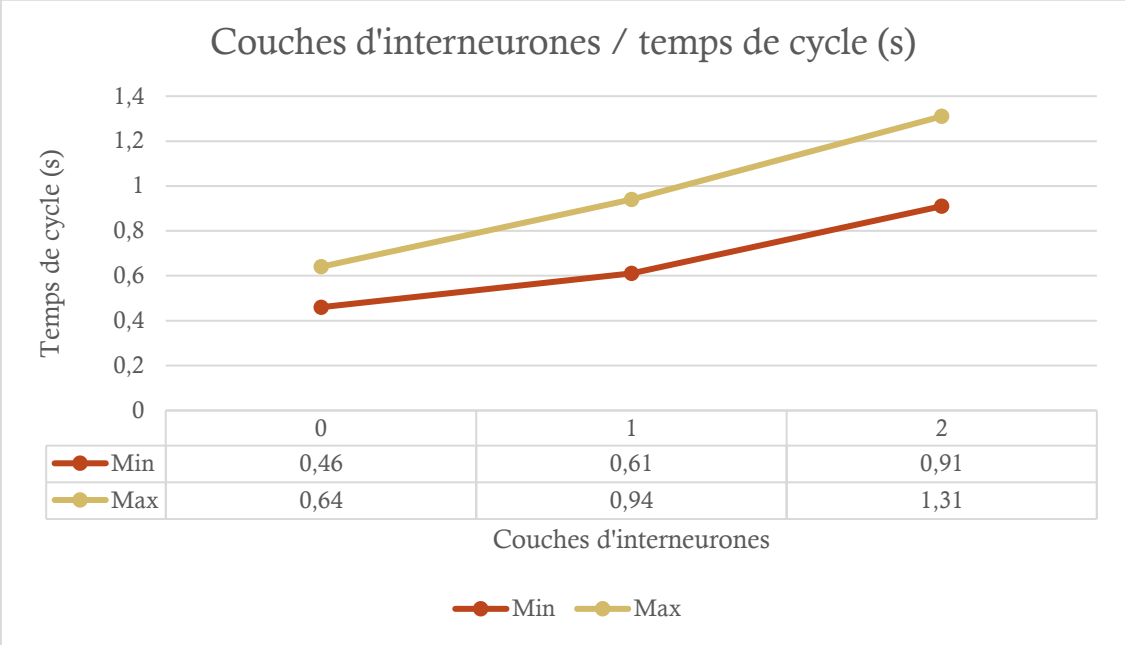


Figure 4.3: Impact du nombre de couches d'interneurones sur le temps de cycle

La figure 4.3 illustre l'impact du nombre de couches d'interneurones sur le temps de cycle. Plus le RND est profond, plus il est demandant en termes de calculs.

La figure 4.4 montre l'impact du nombre de neurones moteurs sur le temps de cycle. Les droites *Min* et *Max* demeurent relativement parallèles. Chacun des neurones moteurs sont liés à un *topic*, mais ceux-ci sont occupés de la même façon, que les neurones moteurs déchargent ou non. Si ceux-ci ne déchargent pas, le *topic* reçoit des valeurs 0. Le coût est le même et c'est pourquoi les droites sont relativement parallèles.

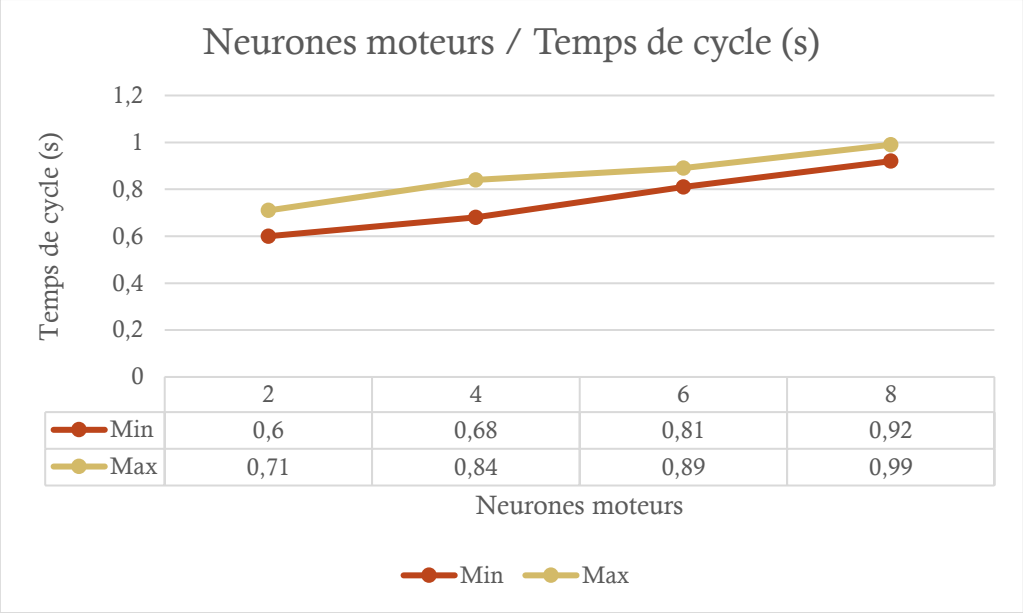


Figure 4.4: Impact du nombre de neurones moteurs sur le temps de cycle

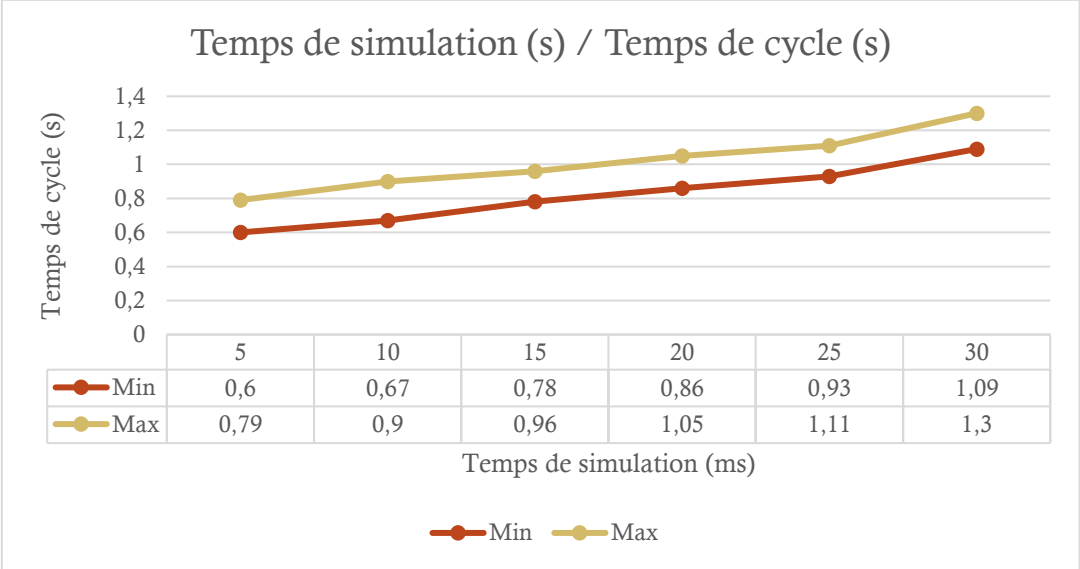
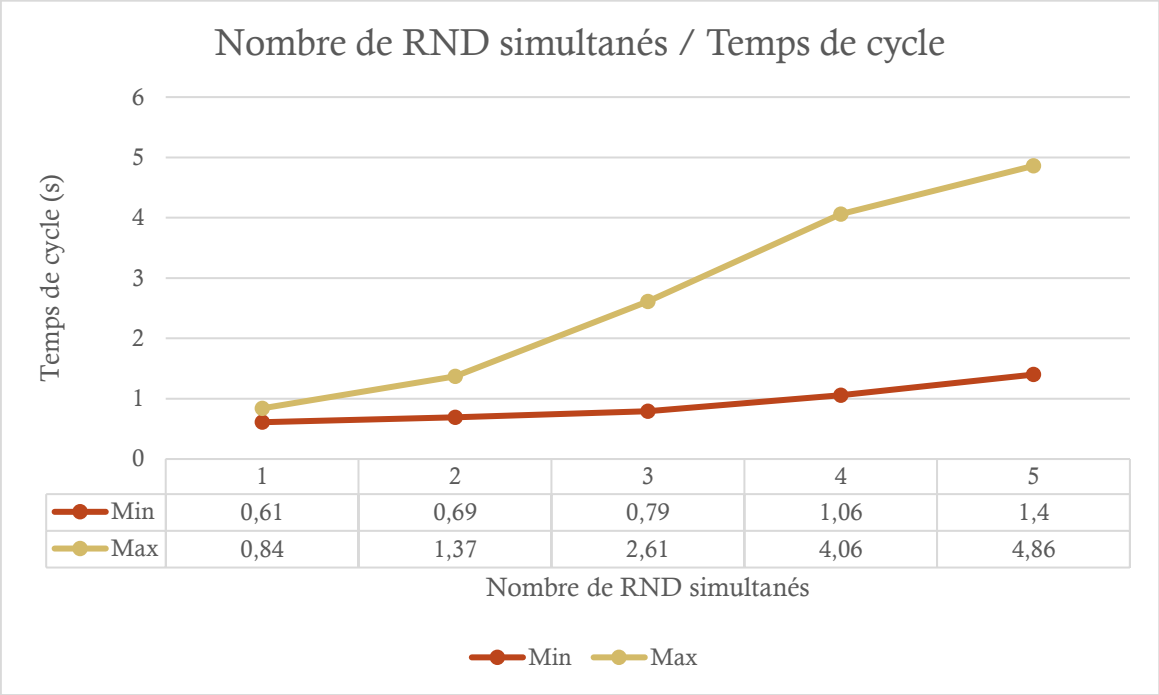


Figure 4.5: Impact du temps de simulation sur le temps de cycle

Varier le temps de simulation affecte de manière importante le RND : il faut utiliser le temps minimal permettant aux potentiels d'action de se produire à l'intérieur du RND. La figure 4.5 montre que le temps de cycle varie linéairement avec le temps de simulation, tant pour la valeur maximale que minimale. Il serait possible de diminuer la valeur sous les 5 ms. Théoriquement,

il est possible de diminuer le temps de simulation tant que le RND dispose d'un temps suffisant pour propager les impulsions jusqu'aux neurones moteurs. Cette valeur minimale requise varie selon l'architecture du RND et le taux de stimulation des neurones sensoriels. Si la valeur de temps de cycle n'est pas suffisamment élevée, les neurones moteurs ne produisent pas de potentiel d'action, rendant le RND inutile. Dans notre exemple, la limite inférieure est située à 0.25 ms de temps de simulation. En deçà de ce temps de simulation, le RND n'a pas suffisamment de temps pour faire les calculs et les neurones moteurs sont toujours au repos. En pratique, il est important de choisir un temps de simulation relativement bas, mais sans trop s'approcher de la limite inférieure, en bas de laquelle le RND devient dysfonctionnel.



4.6: Impact du nombre de RND simultanés sur le temps de cycle

Enfin, la figure 4.6 présente le temps de cycle en fonction du nombre de RND exécutés en parallèle. Pour ce test, plusieurs instances (respectivement d'une à cinq instances) de l'exemple `JoystickTest` sont exécutés simultanément. C'est par ce test qu'est démontré la possibilité d'exécution simultanée de plusieurs RND. Les RND correspondent à la configuration de la figure 3.2. Les résultats sont passablement linéaires, mais les courbes *Min* et le *Max* divergent, puisque le *Max* utilise un plus grand nombre de ressources. La raison de cette divergence est

qu'il y a un coût autre que la boucle principale de l'algorithme du RND lui-même. L'algorithme de RND doit utiliser des fonctions de rappels (*callback*) dès que de nouvelles valeurs arrivent des senseurs. Ceci a un coût en termes de temps de traitement, et c'est ce coût qui est observé dans le graphique.

Ces tests mettent en évidence des temps de cycle qui varient en fonction du nombre de neurones, comme le montre la figure 4.7. Une régression a été calculée à partir des nuages de points générés par les valeurs minimales et maximales des temps de cycle. Les régressions des valeurs minimales et maximales sont linéaires. L'écart R^2 est expliqué par la différente nature des neurones testés. Le coût de certains types de neurones peut être légèrement plus élevé que d'autres, dû à des accès (en lecture ou en écriture) à certains canaux de communication ROS.

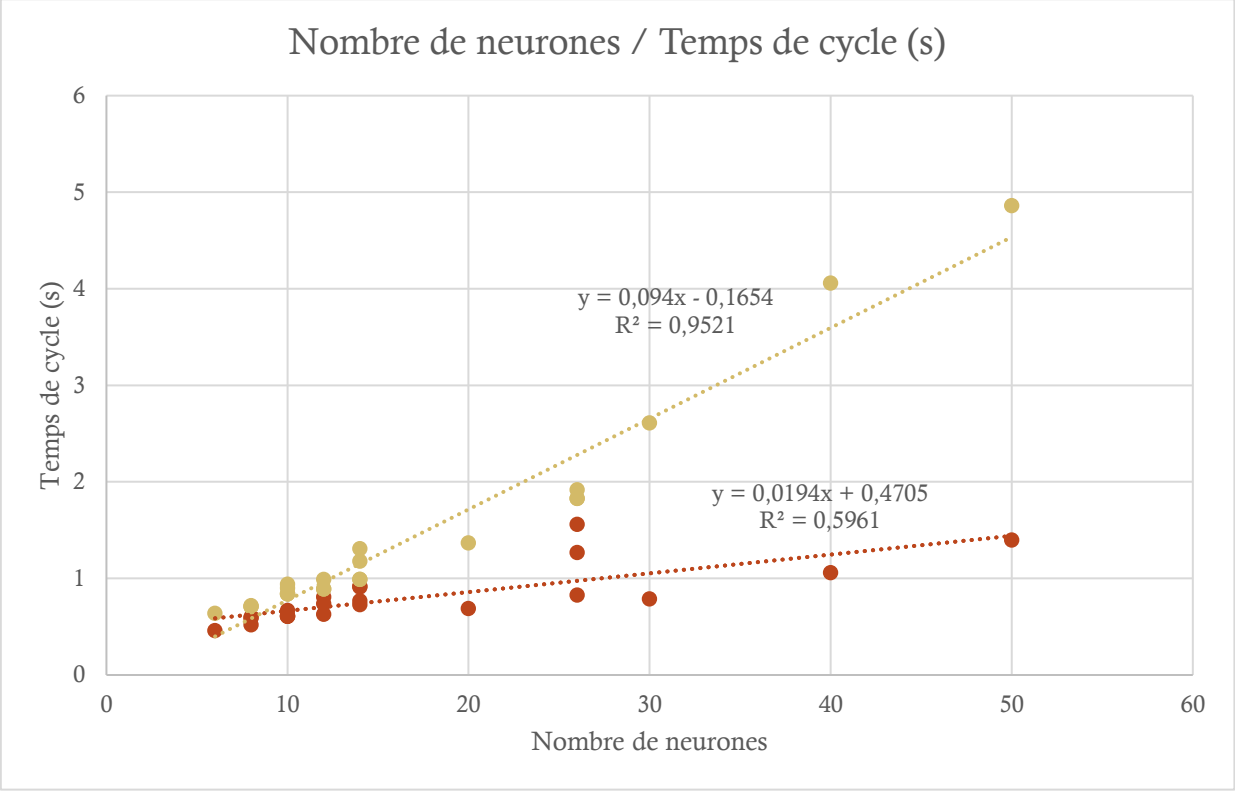


Figure 4.7: Impact du nombre de neurones total d'un RND sur le temps de cycle

Ces régressions suggèrent que le temps de cycle (y) pour un RND s'estime en fonction du nombre de neurones (x) par les équations (4.1) comme minimum et (4.2) comme maximum.

$$y = 0{,}0194x - 0{,}4705 \quad (4.1)$$

$$y = 0{,}094x - 0{,}1654 \quad (4.2)$$

4.2 Test d'un RND qui gère la navigation du robot Spike

Figure 4.8: Le robot Spike

Pour valider la librairie ROS-SNN, le robot Spike montré à la figure 4.8 fut conçu. Il est composé de deux Raspberry Pi 3B, un nano processeur présenté à la figure 4.9. Ces Raspberry Pi 3B disposent d'un processeur Broadcom BCM2837 64 bit à quatre cœurs ARM Cortex-A53 cadencé à une vitesse de 1.2 GHz. Ils possèdent une puce WIFI 802.11n et d'une interface Bluetooth 4.1. Ils disposent également de deux ports USB 2.0, d'un port réseau *Fast Ethernet* à 10/100 Mbit/s, d'un port HDMI, d'une prise Jack 3.5 mm, ainsi que d'un port GPIO (*General Purpose Input/Output*) pour les autres entrées et sorties. Leur mémoire vive est de 1 Go et le GPU est de type Broadcom VideoCore IV. Leur puissance nominale est de 800 mA (4 W) et

leur consommation maximale mesurée est de 720 mA. Leur dimension est de 85,60 mm × 53,98 mm × 17 mm. Ils sont alimentés par un adaptateur 2.5 A.

Figure 4.9: Raspberry Pi 3B

Sur chacun des Raspberry Pi 3B est installé un système d'exploitation Raspian Jessy et ROS (version Turtle Jade). Il possède une caméra de profondeur (Kinect) et un microphone USB standard. Un écran HDMI y est installé, ainsi que deux haut-parleurs, sur une plateforme mobile Pioneer 2 qui lui permet de se déplacer. Il est également équipé d'un routeur sans-fil qui permet la communication en LAN entre les deux Raspberry Pi 3B. Finalement, une manette de jeu USB y est installée afin d'assurer le téléguidage du robot, ainsi que pour permettre des tests de base sur le RND.

La caméra de profondeur possède toutes les fonctionnalités de base prévue par la librairie `Libfreenect [40]`, qui agit comme pilote de périphérique. La caméra de profondeur est aussi utilisée par la librairie nommée `depthimage_to_laserscan [41]`. Cette dernière transforme le signal de la caméra de profondeur en émulation de signal de détection de proximité laser (*Laser Range Finder*, LRF). La librairie de reconnaissance vocale utilisée est `speech_recog [42]` et la librairie de synthèse vocale est `espeak [43]`.

Le robot Spike[5] peut être contrôlé par des commandes vocales ou par la manette de jeu. Les modules de traitement pour la reconnaissance vocale et la voix de synthèse sont utilisés

[5] Le code est disponible à l'adresse suivante : https://github.com/jsdessureault/spike.

directement sans utiliser de RND. Les autres fonctionnalités utilisent un très grand flux de données. En conséquence, la taille du RND et sa charge de traitement associée auraient été trop exigeantes pour un nano processeur de type Raspberry Pi 3B. Seule la navigation du robot utilise un RND. Le but est de faire naviguer le robot Spike navigue selon les instructions vocales suivantes :

- *Avance* pour faire avancer le robot.
- *Arrêter* pour faire arrêter le robot.
- *Gauche* pour que le robot tourne vers la gauche.
- *Droite* pour que le robot tourne vers la droite.

La communication entre les nœuds se fait selon en utilisant les canaux de communication ROS. Les valeurs exactes de vitesses et de rotation lors de virages sont spécifiées à la section 4.3. Il faut aussi qu'il soit possible de téléopérer le robot avec une manette de jeu, selon les commandes illustrées à la figure 4.10.

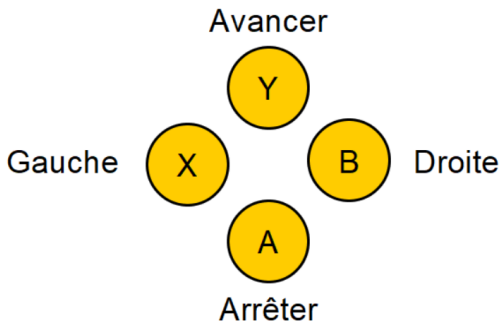


Figure 4.10: Commandes sur les boutons de la manette de jeu

La figure 4.11 montre l'architecture du robot Spike, avec son RND de navigation. La caméra de profondeur, le microphone et la manette de jeu envoient des signaux saisis dans l'environnement du robot. Ces signaux sont encodés sous forme de canaux de communication ROS, et envoyés aux comportements standards ou au RND de navigation, selon le cas. Les comportements standards sont faits de librairies traditionnelles. Par exemple, la reconnaissance vocale utilise la librairie nommée `Speech_recog [42]`. Le RND de navigation contient cinq neurones sensoriels : un pour chaque instruction (Avance, Arrêter, Gauche, Droite), ainsi qu'un neurone

pour le signal qui provient de la vision artificielle (le signal provenant de la caméra de profondeur, converti en LRF). Ils sont tous liés par des synapses à leur équivalent en termes de neurones moteurs, comme le montre la figure 4.12. Chaque neurone sensoriel envoie un signal positif à son équivalent moteur. Le neurone sensoriel lié à l'instruction d'arrêt envoie aussi une rétroaction positive à son neurone moteur, mais vient également inhiber tous les autres instructions (Avance, Gauche et Droite). La raison est qu'il est impératif de prioriser l'instruction d'arrêt afin d'éviter les collisions. Le neurone sensoriel lié au LRF est lié à son neurone moteur jumeau ainsi qu'au neurone d'arrêt. Il vise qu'à ajuster la vitesse du robot. Plus un obstacle est près du robot, moins la vitesse est grande. Un objet trop près du robot contraint le robot à l'arrêt total. De façon plus spécifique, les données représentant l'entièreté de l'image tirée de la caméra de profondeur est convertie en une image à une seule dimension représentant une distance, tel un signal de LRF. La librairie `depthimage_to_laserscan` convertie cette image à deux dimensions en image à une seule dimension. Le rôle du comportement `behavior_laser2snn` est particulièrement important pour la suite. Celui-ci utilise le tier des signaux les plus au centre du signal LRF émulé à l'étape précédente. Il divise la distance (exprimée en mètres) de l'objet le plus près du robot par la distance maximale de détection. Le résultat donne une valeur normalisée entre 0 et 1. Cette valeur est ensuite soustraite à 1. Une autre valeur normalisée entre 0 et 1 est obtenue de cette soustraction. Plus l'obstacle est près du robot, plus la valeur normalisée finale est élevée. Ce nombre est la valeur envoyée au neurone sensoriel de détection laser, qui est aussi lié au neurone d'arrêt.

Toutes les entrées se font à l'aide du nœud `Convertisseur d'entrée`. L'architecture du RND est déclarée dans le fichier `snn_spike.xml`. La rétroaction du RND se fait par les neurones moteurs. Ceux-ci envoient un signal aux actionneurs (base mobile motorisée, haut-parleurs et écran), afin d'avoir un retour sur l'environnement. Le robot peut également être branché par connexion sans fil à Internet ou à une grappe de serveur pour plus de puissance. Ces fonctionnalités de connexion à une grappe de serveurs sont déjà implémentées par ROS et s'utilisent de façon transparente par la librairie ROS-SNN.

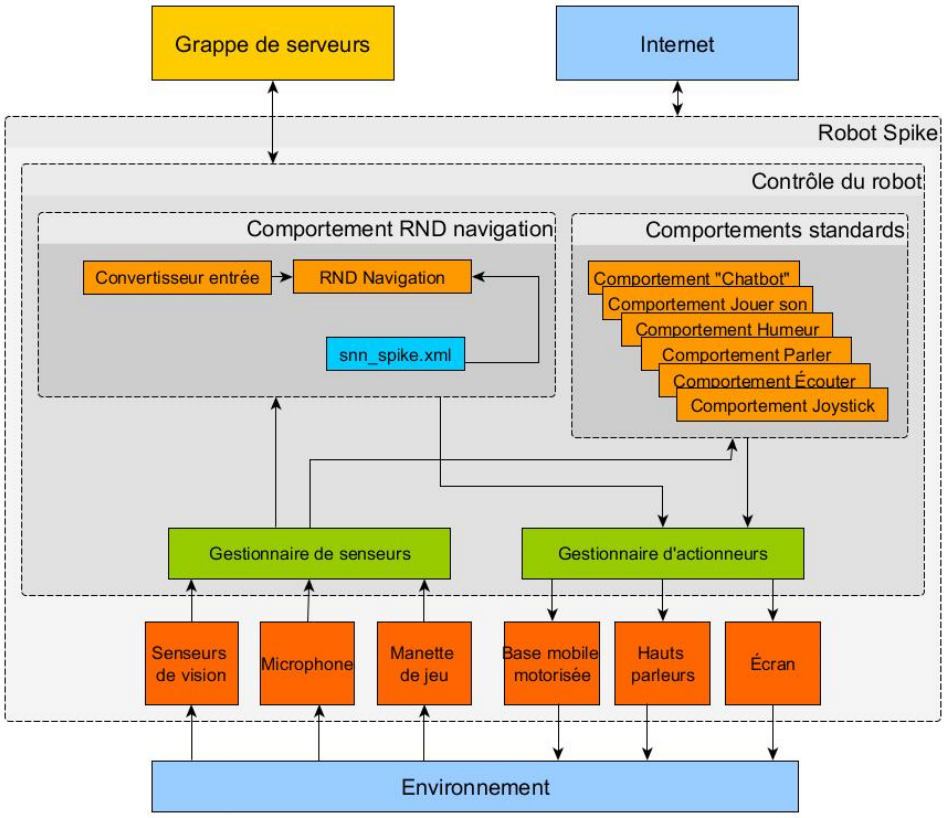


Figure 4.11 : Architecture décisionnelle du robot Spike

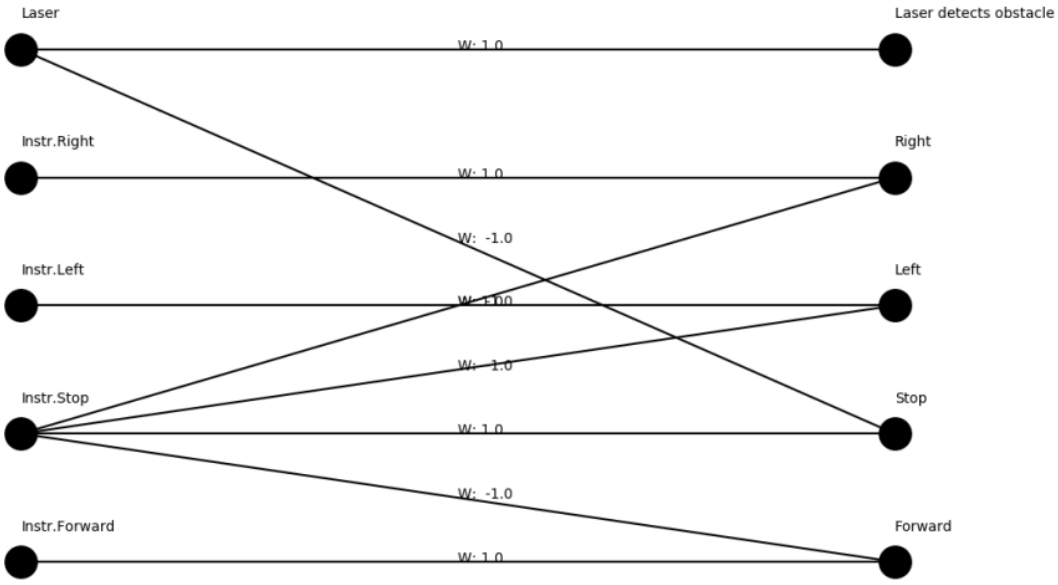


Figure 4.12: Architecture du RND de navigation du robot Spike

Le code pour configurer le RND est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<SNN
   name="spike"
   synapse_delay="0.1"
   input_drive_current="0.1"
   tau="10"
   threshold="v>=0.9"
   reset="v=0"
   refractory="5"
   sim_lenght="5">
   <layer type="sensory" name="sensory">
      <neuron id="0">Instr.Forward</neuron>
      <neuron id="1">Instr.Stop</neuron>
      <neuron id="2">Instr.Left</neuron>
      <neuron id="3">Instr.Right</neuron>
      <neuron id="4">Laser</neuron>
   </layer>
   <layer type="motor" name="motor">
      <neuron id="0" synapse="0, 1" layer="sensory " weight="1.0, -
1.0">Forward</neuron>
      <neuron id="1" synapse="1, 4" layer="sensory" weight="1.0, -
1.0">Stop</neuron>
      <neuron id="2" synapse="2, 1" layer="sensory" weight="1.0, -
1.0">Left</neuron>
      <neuron id="3" synapse="3, 1" layer="sensory" weight="1.0, -
1.0">Right</neuron>
      <neuron id="4" synapse="4" layer="sensory" weight="1.0">Laser detects
obstacle</neuron>
   </layer>
</SNN>
```

Figure 4.13 : Code XML de l'architecture du RND de navigation de Spike

Pour lancer les tests sur le robot, il faut entrer les commandes suivantes :

- Sur le nano ordinateur principal (192.168.0.3) :
    - `roscore` (pour lancer ROS);
    - `roslaunch spike behavior_spike_joy.launch` (pour lancer le RND de navigation);
    - `rosrun spike behavior_navigation.py` (pour lancer le comportement de navigation);
    - `rosrun spike behavior_laser2snn.py` (pour émuler un laser à partir du signal de la caméra de profondeur).

- Sur le nano ordinateur secondaire (192.168.0.1) :
    - `sh go_joy.sh` (pour lancer la manette de jeu);
    - `sh go_kinect.sh` (pour lancer la caméra de profondeur);
    - `sh go_rosaria.sh` (pour lancer la base mobile).

4.3 Résultats

Bien qu'il soit possible de lui faire parvenir les commandes de façon vocale, le test consiste à lui passer les commandes à l'aide de la manette de jeu, afin de s'assurer de la qualité du signal d'entrée du RND. La reconnaissance vocale est plus sujette à erreur et moins fiable qu'une manette de jeu. Le tableau 4.1 présente la validation de chacune des instructions de navigation reconnue par le robot, ainsi que sa capacité à détecter les obstacles.

Tableau 4.1 : Validation des commandes versus actions du robot

Commandes	**Actions**
Avancer	Le robot avance à une vitesse constante de 0.6 m/s. Il continue d'avancer à cette vitesse tant qu'il ne reçoit pas de signal d'arrêter et qu'il n'y a pas d'obstacle devant lui.
Arrêter	Le robot s'immobilise, s'il n'était pas déjà immobile.
Gauche	Le robot (à l'arrêt ou en mouvement) tourne à gauche à une vitesse de -0.15 rad/s (-8.59 deg/s). Si le robot avait déjà un mouvement avant, il continue à avancer tout en tournant. Si le robot était immobile, il fait une rotation sur lui-même.
Droite	Le robot (à l'arrêt ou en mouvement) tourne à droite à une vitesse de 0.15 rad/s (8.59 deg/s). Si le robot avait déjà un mouvement avant, il continue à avancer tout en tournant. Si le robot était immobile, il fait une rotation sur lui-même.
Détection d'un obstacle	Le robot en marche ralentit en fonction de la distance de l'objet détecté. Il s'immobilise lorsqu'il se trouve à 1 mètre de l'obstacle détecté.

Le temps d'exécution par cycle du RND varie de 0.7 sec lorsque les senseurs ne sont pas stimulés, à 1.22 sec lorsque les senseurs sont stimulés à pleine capacité. Ceci démontre que ROS-SNN est fonctionnel dans un petit système de navigation sur un robot mobile muni d'un Raspberry Pi 3B. Les tableaux 4.2 et 4.3 présentent les charges de travail sur les deux Raspberry Pi 3B. Le tableau 4.2 présente la répartition de la charge de traitement sur le premier nano processeur. Les pourcentages de CPU affichés peuvent être plus grands que 100 %. La raison est que cette valeur représente le ratio d'utilisation de la tâche par rapport à un seul des quatre cœurs d'un Raspberry Pi 3B. Une valeur supérieure à 100 % signifie qu'il faut plus d'un cœur pour exécuter la tâche. Chaque Raspberry Pi 3B a donc une capacité totale de 400 % (quatre cœurs à 100 %). La capacité totale de la mémoire vive est de 1 Go.

Tableau 4.2 : Répartition de la charge de traitement sur le premier Raspberry Pi 3B

Tâches	**%CPU**	**%Mémoire**
RND de navigation	100,3 %	9,9 %
Conversion des entrées	0,7 %	3,7 %
Agent conversationnel	0,3 %	2,8 %
Reconnaissance vocale	1,3 %	3,2 %
Synthèse vocale	0,3 %	3,2 %
ROS (*roscore*)	4,7 %	1,1 %
Affichage du visage du robot	91,7 %	8,6 %

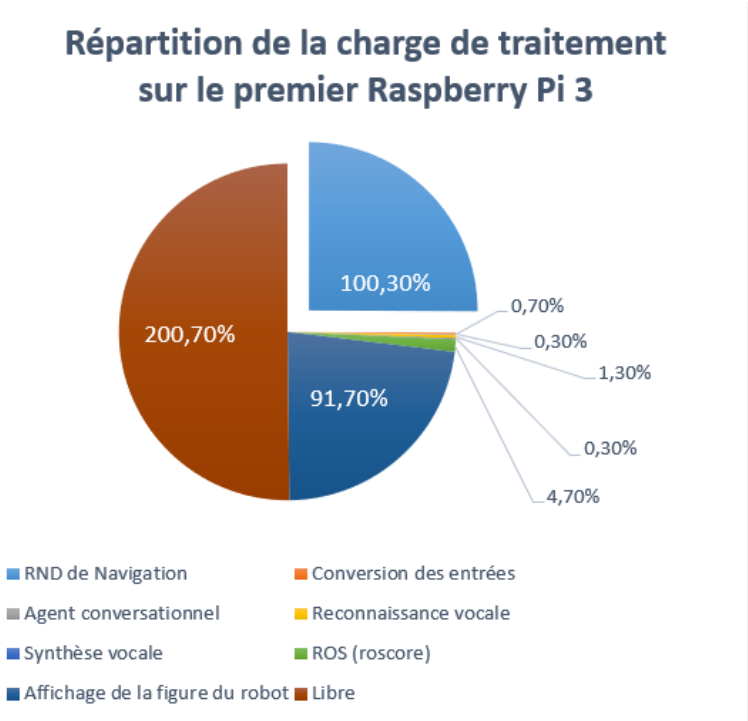


Figure 4.14 : Répartition de la charge de traitement sur le premier Raspberry Pi 3B

La figure 4.14 montre qu'environ le quart de la puissance de calcul est utilisé pour le traitement du RND de navigation. Environ la moitié de la puissance de calcul n'est pas utilisée. Cette puissance potentielle pourra servir dans le cas ou un plus grand RND serait nécessaire, ou si ce RND était très stimulé par de nombreuses entrées venant des senseurs.

La figure 4.15 montre la répartition de la mémoire sur le premier Raspberry Pi 3B du robot. Un petit RND utilise approximativement 10 % de la mémoire disponible. La mémoire s'avère donc suffisante pour un réseau de neurones de petite taille comme celui de la navigation d'un robot. Même en exécutant d'autres processus (comme la reconnaissance vocale, la voix de synthèse et un agent conversationnel), il reste plus des deux-tiers (67,5 %) de la mémoire vive disponible.

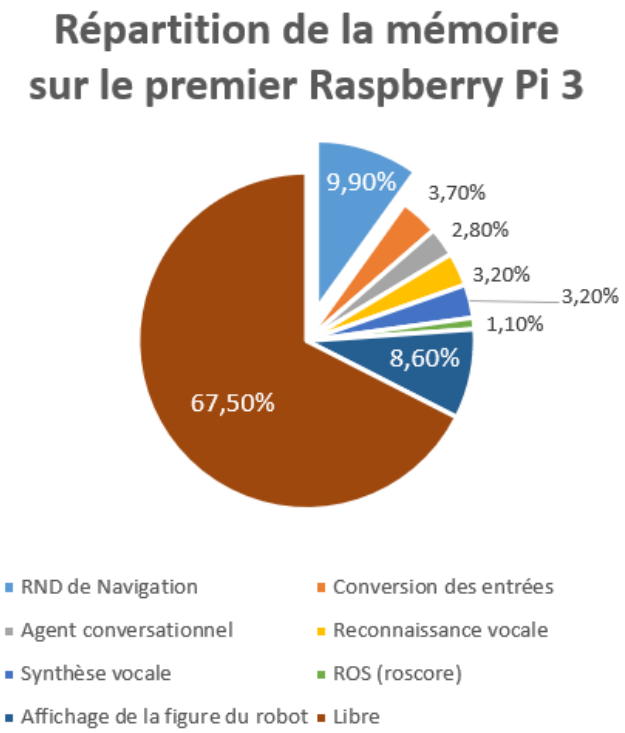


Figure 4.15 : Répartition de la mémoire sur le premier Raspberry Pi 3B

Le tableau 4.3 présente la répartition de la charge de traitement sur le deuxième Raspberry Pi 3B. Ce deuxième Raspberry Pi 3B exécute les fonctions de vision et de mobilité du robot.

Tableau 4.3 : Répartition de la charge de traitement sur le deuxième Raspberry Pi 3B.

Tâches	**%CPU**	**%Mémoire**
Plateforme mobile (Rosaria)	6,7 %	1,3 %
Caméra de profondeur (Kinect)	13,6 %	28,3 %
Manette de jeu	0,3 %	0,8 %
Conversion de la caméra de profondeur en signal laser (*depthimage to laserscan*)	0,3 %	3,1 %

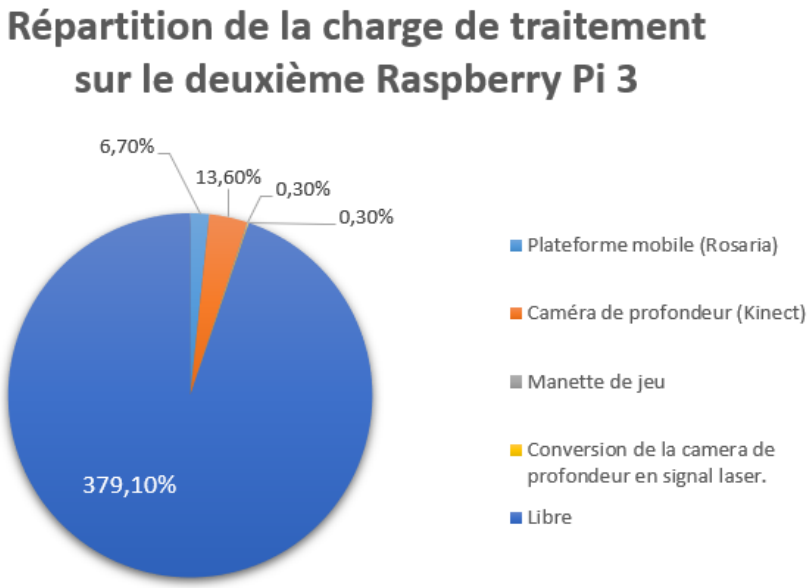


Figure 4.16 : Répartition de la charge de traitement sur le deuxième Raspberry Pi 3B

La figure 4.16 illustre que le deuxième nano processeur n'est pas surchargé même en exécutant des fonctions importantes de la mobilité du robot. Le nœud qui demande le plus de traitement est celui de la caméra de profondeur (13,6 % de la charge d'un cœur). Cependant, la lecture de ces données a été faite lorsque le robot était immobile. Le mouvement du robot génère des entrées supplémentaires des senseurs. Il peut donc s'avérer utile de garder une marge de manœuvre afin de pouvoir subvenir à une demande en puissance de calcul soudainement plus forte. Toutefois, le deuxième nano processeur est surtout important pour sa contribution en termes de mémoire. La figure 4.17 montre qu'environ le tier de la mémoire totale est utilisée lorsque le robot est au repos. Tout comme c'était le cas pour le premier Raspberry Pi 3B, il est important de ne pas surcharger la mémoire afin d'assurer un fonctionnement fluide des diverses fonctionnalités du robot. L'utilisation des cœurs des nano processeurs et de la mémoire peuvent varier considérablement.

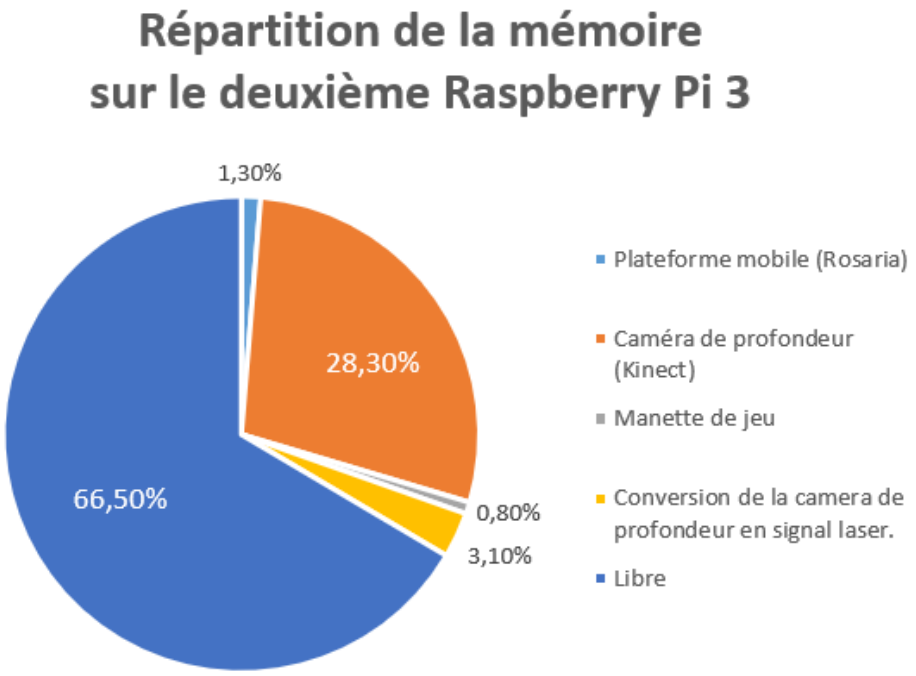


Figure 4.17 : Répartition de la mémoire sur le deuxième Raspberry Pi 3B

Bien que le système pourrait s'exécuter sur un seul Raspberry Pi 3B, la répartition sur deux nano-ordinateurs est préférable. Puisque les Raspberry Pi 3B sont très peu dispendieux, et puisque ROS peut s'exécuter très facilement sur plusieurs machines, il est avantageux de laisser de la marge de manœuvre (tant au point de vue puissance de calcul qu'au point de vue de la mémoire) au système en exécution.

Enfin, afin de tester les limites du système, plusieurs instances du même RND de navigation ont été créées simultanément. Avec six instances simultanées, tout fonctionne et le temps de cycle est de 1,35 sec. En instanciant une septième instance toutefois, les ressources ne sont plus suffisantes : dès l'instanciation, le cycle de traitement passe à 7,83 sec avant que le système ne s'arrête définitivement.

CHAPITRE 5 ANALYSE

Les résultats du chapitre 4 suggèrent qu'il est possible d'utiliser un RND sous ROS, même sur une plateforme matérielle avec d'aussi faibles capacités qu'un Raspberry Pi 3B, sous certaines conditions :

- Le nombre de neurones et de synapses doit être gardé le plus bas possible. Un réseau de neurones profond va ralentir grandement le traitement jusqu'à ce qu'il devienne impossible de garder le temps de cycle en deçà du seuil défini par l'usager. Il est très difficile de déterminer un temps limite pour l'exécution d'un cycle. Ce temps peut varier selon la nature du problème. Cependant, à la lecture du graphique de la figure 4.7 et de ses équations, il est facile de calculer un nombre de neurones à ne pas dépasser pour s'assurer de demeurer en deçà d'une certaine limite de temps de cycle. Par exemple, pour garder le traitement sous les y=2 sec, l'équation (4.2) donne x=23 neurones.
- Si la plateforme contient des processus demandant des processus lourds en calculs ou en mémoire comme l'analyse d'images RGB-D d'une caméra de profondeur, ils doivent être idéalement placés sur un second nano processeur lié au nano processeur central avec les fonctionnalités de ROS.
- Il faut éviter que les signaux d'entrées ne créent un goulot d'étranglement. Il est préférable de garder les entrées des senseurs au minimum requis afin de maximiser les performances du RND.
- Il est possible d'exécuter plusieurs instances de RND sur un même processeur Raspberry Pi 3B. Mais les conditions ci-haut doivent être respectés, et les équations (4.1) et (4.2) s'appliquent sur le total des neurones des différents RND en exécution.

Dans le cas d'un RND appliqué à la robotique, il faut que le système soit en mesure de garder un temps de cycle relativement bas, selon la nature de la fonction. À la lecture des figures 4.14 à 4.17 ainsi que de la figure 4.7, il est possible de constater que ce n'est ni l'utilisation du microprocesseur ni la capacité de la mémoire qui risque de faire défaut. Il s'agit plutôt du temps de cycle qui risque de devenir trop long, au fur et à mesure que le nombre total de neurones et de synapses augmente, dans tous les RND en cours d'exécution.

Les équations 4.1 et 4.1 demeurent une référence en termes de temps de cycle. Il en revient à l'utilisateur de la librairie à décider si ces temps de cycle sont acceptables en ce qui a trait aux performances du robot.

CHAPITRE 6 CONCLUSION

La librairie ROS-SNN développée au cours des travaux de ce mémoire permet de créer différentes configurations et architectures de RND exécutables simultanément et utilisant un format simple et compatible avec ROS. Elle est aussi dotée d'utilitaires pratiques permettant de visualiser les entrées et sorties ainsi que le diagramme d'architecture du RND. Ces fonctionnalités sont très utiles pour comprendre le fonctionnement du RND.

Ce travail de recherche est une première étape pour rendre plus accessibles les RND aux roboticiens. L'impact sera intéressant dans la mesure où la communauté verra un intérêt à implémenter facilement des RND paramétrables et réutilisables sur des robots. Il faut bien comprendre que l'intérêt derrière l'usage de RND est de trouver un dénominateur commun au mécanisme d'apprentissage et de prise de décision. Les RND s'inspirent du monde animal pour créer une forme d'intelligence plus générale que spécifique. Les RND du futur seront appuyés à court terme par des processeurs graphiques qui permettent un traitement en parallèle, mais aussi à moyen et à long terme avec les processeurs neuromorphes. Ces derniers pourront donner une réelle chance aux RND de se démarquer par rapport à des techniques autres que neuronales.

Dans le futur, certaines améliorations pourraient être apportées à la librairie ROS-SNN. Premièrement, il serait utile de pouvoir entrainer des RND de type STDP (*Spike Timing Dependant Plasticity*) [30] afin d'ajuster les poids synaptiques grâce à un processus d'apprentissage. Pour l'instant, toutes les connexions synaptiques doivent être définies dans le fichier de définition XML. Puisque nous n'utilisons que des RND de petite taille, cette étape peut se faire facilement. Cependant, pour des RND de plus grande taille, cette possibilité d'entrainement STDP devient essentielle. Une autre piste d'amélioration serait de permettre au RND de s'exécuter en parallèle sur l'unité de traitement graphique (GPU). La librairie Brian 2 permet ce traitement, mais la librairie ROS-SNN n'exploite pas ces capacités pour l'instant. Un autre beau défi serait de modifier les algorithmes de ROS-SNN pour permettre le traitement parallèle sur GPU, tout en gardant l'usage de la librairie simple. Bien que ROS-SNN permet l'exécution de plusieurs RND simultanément, il n'utilise pas le GPU, et le traitement n'est pas parallèle à l'intérieur d'un même RND.

Dans sa version actuelle, il peut être compliqué de définir un RND de taille moyenne ou grande dans un fichier XML. Un éditeur convivial serait bienvenu dans la librairie ROS-SNN. Il serait utile de pouvoir générer facilement un fichier XML définissant un RND grâce à un interface utilisateur simple.

Un autre objectif sous-jacent à ROS-SNN, en plus d'offrir un outil pour l'intégration des RND sur différentes plateformes robotiques, est de permettre éventuellement l'utilisation des RND sans pour autant être un expert en la matière. Le souhait est que les RND deviennent une solution de plus en plus intéressante pour le traitement de l'information et la prise de décision en robotique.

BIBLIOGRAPHIE

[1] A. M. Turing, “Computing machinery and intelligence,” *Mind 4,* pp. 433-460, 1950.

[2] A. Hodgkin et A. Huxley, “The components of membrane conductance in the giant axon of Loligo,” *The Journal of Physiology,* n°1116, pp. 473-496, 1952.

[3] D. O. Hebb, *The Organization of Behavior*, New York: Wiley & Sons, 1949.

[4] W. McCulloch et W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *Bulletin of Mathematical Biophysics,* p. 115–133, 1943.

[5] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain,” *Psychological Review,* pp. 65-386, 1958.

[6] J. Hopfield, “Neural networks and physical systems with emergent collective computational abilities,” *Proceedings of the National Academy of Sciences,* vol. 79, n°1 April, pp. 2554-2558, 1982.

[7] D. H. Ackley, G. E. Hinton et T. J. Sejnowski, “A learning algorithm for Boltzmann machines,” *Cognitive Science,* pp. 147-169, 1985.

[8] A. Hodgkin et A. Huxley, “A quantitative description of membrane current and its application to conduction and excitation in nerve,” *The Journal of Physiology,* vol. 117, pp. 500-544, 1952.

[9] E. M. Izhikevich, “Simple model of spiking neurons” *IEEE Transactions on Neural Networks,* vol. 14, n°16, pp. 1569-1572, 2003.

[10] M. Quigley, K. Conley, J. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler et A. Ng, “ROS: an open-source Robot Operating System,” *ICRA Workshop on Open Source Software,* vol. 3, pp. pp. 5-11, 2009.

[11] N. J. Nilsson, “Human-level artificial intelligence? Be serious!” *AI Magazine,* vol. 26, n° 14, p. 68, Winter 2005.

[12] M. L. Minsky et S. Papert, *“Perceptrons: An Introduction to Computational Geometry*,” Cambridge: The MIT Press, 1969.

[13] A. Hodgkin et A. Huxley, “Measurement of current-voltage relations in the membrane of the giant axon of Loligo,” *Journal of Physiology,* n° 1116, pp. 424-448, 1952.

[14] A. Hodgkin et A. Huxley, “Currents carried by sodium and potassium ions through the membrane of the giant axon of Loligo”, *Journal of Physiology,* vol. 116, pp. 449-472, 1952.

[15] L. Lapicque, “Recherches quantitatives sur l'excitation électrique des nerfs traitée comme une polarisation,” *Société de Biologie,* vol. 9, pp. 620-635, 1907.

[16] W. Gerstner et W. M. Kistler, “EPFL - Spiking neuron models” 2002. [En ligne]. http://icwww.epfl.ch/~gerstner/SPNM/node26.html.

[17] R. K. Belew, J. McInerney et N. N. Schraudolph, “*Evolving Networks: Using the Genetic Algorithm with Connectionist Learning*,” Cognitive Computer Science Research Group Computer Science & Engr. Dept. (C-014) Univ. California at San Diego, 1990

[18] S. Juncheng, M. De, G. Zonghua et M. Zhang, “Darwin: A neuromorphic hardware co-processor based on spiking neural networks,” *Science China Information Sciences,* vol. 59, n°12, pp. 1-5, 2016.

[19] A. Trewavas, “Aspects of plant intelligence,” *Annals of Botany,* vol. 92, n°11, pp. 1-20, 2003.

[20] R. Brette et D. F. Goodman, “Simulating spiking neural networks on GPU,” *Network: Computation in Neural Systems,* vol. 23, n°14, pp. 167-182, 2012.

[21] A. K. Fidgeland, E. B. Roesch et M. P. Shanahan, “NeMo: A platform for neural modelling of spiking neurons using GPUs,” *20th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, Boston, MA, 2009.

[22] J.-S. Seo, B. Brezzo et L. Yong, “A 45nm CMOS neuromorphic chip with a scalable architecture for learning in networks of spiking neurons,” *IEEE Custom Integrated Circuits Conference*, 2011.

[23] G. Indiveri, B. Linares-Barranco, R. Legenstein et G. Deligeorgis, “Integration of nanoscale memristor synapses in neuromorphic computing architectures,” *Nanotechnology,* vol. 24, n°138, p. 13, 2013.

[24] J. Hsu, “IBM's new brain,” *IEEE Spectrum,* pp. 17-19, Octobre 2014.

[25] “SpiNNaker briain simulation machine” http://www.artificialbrains.com/spinnaker .

[26] D. Feldmeyer, M. Brecht, F. Helmchen, C. C. Petersen et J. F. Poulet, “Barrel cortex function,” *Progress in Neurobiology,* vol. 103, pp. 3-27, 2012.

[27] Y. Kremer, J.-F. Léger, D. Goodman, R. Brette et L. Bourdieu, “Late emergence of the vibrissa direction selectivity map in the rat barrel cortex,” *JNeurosci - The Journal of Neuroscience,* vol. 103, p. 31 (29), 2011.

[28] “From papers: Kremer et al. 2011 barrel cortex” 2012. [En ligne]. http://brian2.readthedocs.io/en/2.0rc3/examples/frompapers.Kremer_et_al_2011_barrel_cortext.html

[29] A. Cyr, *Intelligence artificielle et robotique bio-inspirée: Modélisation de fonctions d'apprentissage par réseaux de neurones à impulsions,* Faculté des sciences, Département d'informatique, UQAM, Montréal, 2016.

[30] A. Cyr et B. Mounir, “Classical conditioning in different temporal constraints: an STDP learning rule for robots controlled by spiking neural networks,” *Adaptive Behavior,* vol. 20, n°14, pp. 257- 272, 2012.

[31] A. Cyr et B. Mounir, “Habituation: A non-associative learning rule design for spiking neurons and an autonomous mobile robots implementation,” *Bioinspirations & Biomimetics,* vol. 8, p. 17, 2013.

[32] A. Cyr, B. Mounir et F. Thériault, “Operant conditioning: a minimal components requirement in artificial spiking neurons designed for bio-inspired robot's controller,” *Frontiers in Neurorobotics,* vol. 8, n°121, pp. 1-13, 2014.

[33] ROS on MATLAB, http://wiki.ros.org/groovy/Planning/Matlab.

[34] A. Cyr, B. Mounir et P. Poirier, “AI-SIMCOG: A simulator for spiking neurons and multiple animat's behaviours,” *Neural Computing & Applications,* vol. 18, pp. 431-446, 2009.

[35] Genesis, http://www.genesis-sim.org/.

[36] Nest, http://www.nest-simulator.org/.

[37] CARLSim, http://www.socsci.uci.edu/~jkrichma/CARLsim/.

[38] NeMo, http://nemosim.sourceforge.net/index.html.

[39] Brian 2, http://brian2.readthedocs.io/en/stable/index.html.

ANNEXE A – FICHIERS DE DÉMARRAGE DU RND JOYSTICKTEST

Fichier `behavior_JoystickTest.launch`

```
<launch>
   <node name="SNN" pkg="ros_snn" type="SNN.py" output="screen"
respawn="false" required="true">
    <param name="path" value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/>
<!-- path where the trained SNN is saved -->
    <param name="SNNname" value="JoystickTest"/> <!-- Name of the SNN -->
    <param name="verbose" type="bool" value="True"/>  <!-- Will display
process if True -->
    <param name="xml" value="snn_JoystickTest.xml"/> <!-- Name of the xml
file -->
  </node>


  <!-- Remapping topics for sensory (input) neurons.  Quantity must match
"sensory_neurons" parameter -->
  <node name="input_converter" pkg="ros_snn" type="input_converter.py"
output="screen" respawn="false" required="true">
    <param name="SNNname" value="JoystickTest"/>     <!-- Name of the SNN -
->
    <param name="verbose" type="bool" value="True"/> <!-- Will display
process if True -->
    <param name="topics_to_convert" type="int" value="4"/> <!-- Number of
sensory neurons -->
    <param name="path" value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/>
<!-- path where the trained SNN is saved -->

    <param name="input_topic_1" value="/joy"/>
<!-- Topic containing the value of the input neuron -->
    <param name="topic_type_1" value="Joy"/>  <!-- Topic containing the
type of the input neuron topic 1-->
    <param name="input_field_1" value=".axes[1]"/> <!-- Field of the topic
containing the value of the input neuron -->
    <param name="input_min_1" value="-1.0"/>  <!-- Minimum value of the
MinMax function -->
    <param name="input_max_1" value="1.0"/>   <!-- Minimum value of the
MinMax function -->
    <param name="stringfile_1" value=""/>     <!-- Definition of the strings
(if defined) -->

    <param name="input_topic_2" value="/joy"/>
<!-- Topic containing the value of the input neuron -->
    <param name="topic_type_2" value="Joy"/>   <!-- Topic containing the
type of the input neuron topic 1-->
    <param name="input_field_2" value=".axes[0]"/> <!-- Field of the topic
containing the value of the input neuron -->
    <param name="input_min_2" value="-1.0"/>  <!-- Minimum value of the
MinMax function -->
    <param name="input_max_2" value="1.0"/>   <!-- Minimum value of the
MinMax function -->
```

```
    <param name="stringfile_2" value=""/>     <!-- Definition of the strings
(if defined) -->

    <param name="input_topic_3" value="/joy"/>
<!-- Topic containing the value of the input neuron -->
    <param name="topic_type_3" value="Joy"/>   <!-- Topic containing the
type of the input neuron topic 1-->
    <param name="input_field_3" value=".axes[4]"/>  <!-- Field of the topic
containing the value of the input neuron -->
    <param name="input_min_3" value="-1.0"/>  <!-- Minimum value of the
MinMax function -->
    <param name="input_max_3" value="1.0"/>   <!-- Minimum value of the
MinMax function -->
    <param name="stringfile_3" value=""/>     <!-- Definition of the strings
(if defined) -->

    <param name="input_topic_4" value="/joy"/>
<!-- Topic containing the value of the input neuron -->
    <param name="topic_type_4" value="Joy"/>  <!-- Topic containing the
type of the input neuron topic 1-->
    <param name="input_field_4" value=".axes[3]"/>    <!-- Field of the
topic containing the value of the input neuron -->
    <param name="input_min_4" value="-1.0"/>  <!-- Minimum value of the
MinMax function -->
    <param name="input_max_4" value="1.0"/>   <!-- Minimum value of the
MinMax function -->
    <param name="stringfile_4" value=""/> <!-- Definition of the strings
(if defined) -->

  </node>
  <!--   <remap from="/joy" to="/topic_JoystickTest"> -->
</launch>
```

Fichier `behavior_stringtest.launch`

```
<launch>
   <node name="SNN" pkg="ros_snn" type="SNN.py" output="screen"
respawn="false" required="true">
    <param name="path" value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/>
<!-- path where the trained SNN is saved -->
    <param name="SNNname" value="stringtest"/>  <!-- Name of the SNN -->
    <param name="verbose" type="bool" value="True"/> <!-- Will display
process if True -->
    <param name="xml" value="snn_stringtest.xml"/>   <!-- Name of the xml
file -->
  </node>


  <!-- Remapping topics for sensory (input) neurons.  Quantity must match
"sensory_neurons" parameter -->
  <node name="input_converter" pkg="ros_snn" type="input_converter.py"
output="screen" respawn="false" required="true">
    <param name="SNNname" value="stringtest"/>
<!-- Name of the SNN -->
```

```
    <param name="verbose" type="bool" value="True"/>   <!-- Will display
process if True -->
    <param name="topics_to_convert" type="int" value="1"/>   <!-- Number of
sensory neurons -->
    <param name="path" value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/>
<!-- path where the trained SNN is saved -->

    <param name="input_topic_1" value="/strcommand"/>
<!-- Topic containing the value of the input neuron -->
    <param name="topic_type_1" value="String"/>  <!-- Topic containing the
type of the input neuron topic 1-->
    <param name="input_field_1" value=".data"/> <!-- Field of the topic
containing the value of the input neuron -->
    <param name="input_min_1" value="-1.0"/>  <!-- Minimum value of the
MinMax function -->
    <param name="input_max_1" value="1.0"/>   <!-- Minimum value of the
MinMax function -->
    <param name="stringfile_1" value="string2snn.csv"/> <!-- Definition of
the strings (if defined) -->


  </node>
  <!--   <remap from="/joy" to="/topic_JoystickTest"> -->
</launch>
```

Fichier `plot_JoystickTest.launch`

```
<launch>
        <node name="plot_volts" pkg="ros_snn" type="plot_volts.py"
output="screen" respawn="false">
                <param name="path"
value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/>  <!-- path where the
trained SNN is saved -->
                <param name="xml" value="snn_JoystickTest.xml"/>   <!-- Name
of the xml file -->
                <param name="SNNname" value="JoystickTest"/>
        </node>
        <node name="plot_spikes" pkg="ros_snn" type="plot_spikes.py"
output="screen" respawn="false">
                <param name="SNNname" value="JoystickTest"/>
                <param name="path"
value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/>  <!-- path where the
trained SNN is saved -->
                <param name="xml" value="snn_JoystickTest.xml"/>   <!-- Name
of the xml file -->
        </node>
        <node name="plot_input" pkg="ros_snn" type="plot_input.py"
output="screen" respawn="false">
                <param name="path"
value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/>  <!-- path where the
trained SNN is saved -->
                <param name="xml" value="snn_JoystickTest.xml"/> <!-- Name
of the xml file -->
                <param name="SNNname" value="JoystickTest"/>
        </node>
```

```
        <node name="plot_connectivity" pkg="ros_snn"
type="plot_connectivity.py" output="screen" respawn="false"
required="true">
                <param name="path"
value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/>  <!-- path where the
trained SNN is saved -->
                    <param name="SNNname" value="JoystickTest"/>      <!--
Name of the SNN -->
                    <param name="xml" value="snn_JoystickTest.xml"/>  <!--
Name of the xml file -->
        </node>

</launch>
```

Fichier `plot_stringtest.launch`

```
<launch>
      <node name="plot_volts" pkg="ros_snn" type="plot_volts.py"
output="screen" respawn="false">

                <param name="path"
value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/>  <!-- path where the
trained SNN is saved -->
                <param name="xml" value="snn_stringtest.xml"/>  <!-- Name of
the xml file -->
                <param name="SNNname" value="stringtest"/>
        </node>
        <node name="plot_spikes" pkg="ros_snn" type="plot_spikes.py"
output="screen" respawn="false">
                <param name="SNNname" value="stringtest"/>
                <param name="path"
value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/>  <!-- path where the
trained SNN is saved -->
                <param name="xml" value="snn_stringtest.xml"/> <!-- Name of
the xml file -->
        </node>
        <node name="plot_input" pkg="ros_snn" type="plot_input.py"
output="screen" respawn="false">
                <param name="path"
value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/> <!-- path where the
trained SNN is saved -->
                <param name="xml" value="snn_stringtest.xml"/> Name of the
xml file -->
                <param name="SNNname" value="stringtest"/>
        </node>
        <node name="plot_connectivity" pkg="ros_snn"
type="plot_connectivity.py" output="screen" respawn="false"
required="true">
                <param name="path"
value="/home/pi/ros_catkin_ws/src/ros_snn/snn/"/>     <!-- path where the
trained SNN is saved -->
                    <param name="SNNname" value="stringtest"/> <!-- Name of
the SNN -->
                    <param name="xml" value="snn_stringtest.xml"/> <!-- Name
of the xml file -->
        </node>

</launch>
```

ANNEXE B – DÉCLARATION DU RND JOYSTICKTEST DANS UN FICHIER XML ET CSV

Fichier `snn_JoystickTest.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<SNN
        name="snn_JoystickTest"
        realtime_limit="0.75"
        synapse_delay="1.0"
        input_drive_current="0.1"
        tau="10"
        threshold="v>0.8"
        reset="v=0"
        refractory="5"
        sim_lenght="5">

        <layer type="sensory" name="sensory">
                <neuron id="0">Axe 1 Up</neuron>
                <neuron id="1">Axe 1 Down</neuron>
                <neuron id="2">Axe 2 Up</neuron>
                <neuron id="3">Axe 2 Down</neuron>
        </layer>
        <layer type="inter" name="inter">
                <neuron id="0" synapse="0" layer="sensory"
weight="1.0">Inter 0</neuron>
                <neuron id="1" synapse="1" layer="sensory"
weight="1.0">Inter 1</neuron>
                <neuron id="2" synapse="2" layer="sensory"
weight="1.0">Inter 2</neuron>
                <neuron id="3" synapse="3" layer="sensory"
weight="1.0">Inter 3</neuron>
        </layer>
        <layer type="motor" name="motor">
                <neuron id="0"  synapse="0, 2" layer="inter" weight="0.5,
0.5">Motor 1</neuron>
                <neuron id="1"  synapse="1, 3" layer="inter" weight="0.5,
0.5">Motor 2</neuron>
        </layer>

</SNN>
```

Fichier `string2snn.csv`

```
string,voltage
Red, 0.1
Green, 0.5
Blue, 1.0
```

ANNEXE C – CODE DE LA LIBRAIRIE ROS-SNN

Fichier SNN.py

```
#!/usr/bin/env python

'''
Filename: SNN.py
Author: Jean-Sebastien Dessureault
Date created: 01/06/2016
Python version 2.7
'''
from brian2 import *
prefs.codegen.target = "cython"

import os
import sys
import rospy
import numpy as np
from std_msgs.msg import String, Float32, Float32MultiArray, Int16
import time
from lxml import etree

node = "node_SNN"
rospy.init_node(node, anonymous=True)
rospy.loginfo(node)

# Retriving parameters from launcher
SNNname = rospy.get_param("~SNNname")
verbose = rospy.get_param("~verbose")
pathSNN = rospy.get_param("~path")
xml = rospy.get_param("~xml")

# output topics
topics_motor_volts = []
topics_motor_spikes = []

# Constants
SENSORY_LAYER = 0             # input layer index
INTER_LAYER = 1
MOTOR_LAYER = 2    # Motor layer index

# Global variables
sensory_neurons = 0
inter_neurons = 0
motor_neurons = 0
synapse_delay = 0
input_drive_current = 0
tau = 0*ms
threshold_value = 0
reset_value = 0
refractory_value = 0* ms
simulation_lenght_int = 0
simulation_lenght_ms = 0* ms
realtime_limit = 50
```

```
# Global variable that receives the frames from the topic.
frames_in = []
time_frame = []
MAX_TIME_FRAME = 25
first_frame = True

# Neurons and synapses vectors
neurons = []                # Array of neuronGroup
synapses = []               # Array of synapses

equation = "dv/dt = (I - v)/tau : 1 (unless refractory) I : 1"

# Test parameters validity
#useXML = True
exitSNN = False
if xml == "":
    rospy.loginfo("Must specify a XML name (XML) in the launcher.")
    exitSNN = True
if SNNname == "":
    rospy.loginfo("Must specify a SNN name (SNNname) in the launcher.")
    exitSNN = True
if pathSNN == "":
    rospy.loginfo("Must specify a path for the SNN (pathSNN) in the
launcher.")
    exitSNN = True
try:
    xml_file = pathSNN + "xml/" + xml
    tree = etree.parse(xml_file)
except:
    rospy.loginfo("Error in XML file: " + xml_file)
    exitSNN = True
# If there is at least one error in the previous test, then exit.
if exitSNN == True:
    sys.exit(1)

def Assing_XML():
    global tree, MOTOR_LAYER, sensory_neurons, inter_neurons, inter_layers,
motor_neurons, synapse_delay, input_drive_current, tau, threshold_value,
reset_value, refractory_value, simulation_lenght_int,simulation_lenght_ms,
realtime_limit

    for rnd in tree.xpath("/SNN"):
        synapse_delay = rnd.get("synapse_delay")
        realtime_limit = float(rnd.get("realtime_limit"))
        input_drive_current = float(rnd.get("input_drive_current"))
        tau = int(rnd.get("tau")) *ms
        threshold_value = rnd.get("threshold")
        reset_value = rnd.get("reset")
        refractory_value = int(rnd.get("refractory")) * ms
        simulation_lenght_int = float(rnd.get("sim_lenght"))
        simulation_lenght_ms = float(rnd.get("sim_lenght")) * ms

    sensory_neurons=0
    inter_neurons=0
    motor_neurons=0
    inter_layers = 0
```

```
    for neuron in tree.xpath("/SNN/layer/neuron"):
        layer_type = neuron.getparent().get("type")
        if layer_type == "sensory":
            sensory_neurons+=1
        if layer_type == "inter":
            inter_neurons+=1
        if layer_type == "motor":
            motor_neurons+=1


def Display_Parameters():
    # Displaying parameters to console
    global SSNname, xml, verbose, sensory_neurons, inter_neurons,
motor_neurons, inter_layers,  \
         synapse_delay, synapse_condition, input_drive_current, tau,
threshold_value, refractory_value, simulation_lenght_int, pathSNN,
equation, realtime_limit
    rospy.loginfo("----Parameters received from launcher OR XML file:----")
    rospy.loginfo("SNNname: " + SNNname)
    rospy.loginfo("xml: " + xml)
    rospy.loginfo("verbose: " + str(verbose))
    rospy.loginfo("Real time limit: " + str(realtime_limit))
    rospy.loginfo("sensory_neurons: " + str(sensory_neurons))
    rospy.loginfo("motor_neurons: " + str(motor_neurons))
    rospy.loginfo("inter_neurons: " + str(inter_neurons))
    rospy.loginfo("synapse_delay: " + synapse_delay)
    rospy.loginfo("input_drive_current: " + str(input_drive_current))
    rospy.loginfo("tau: " + str(tau))
    rospy.loginfo("threshold: " + str(threshold_value))
    rospy.loginfo("refractory: " + str(refractory_value))
    rospy.loginfo("simulation_lenght: " + str(simulation_lenght_int))
    rospy.loginfo("path: " + pathSNN)
    rospy.loginfo("equation: " + equation)

# Initialize input frames.
def init_frames_in():
    for x in range(0, sensory_neurons):
        frames_in[x] = 0.0

# Callback triggered when there is a new message on the topic.
def callbackReceiveMsgFromTopic(data, sensory_nb):
    #rospy.loginfo("Received in the callback: neuron: %i  dat: %s",
sensory_nb, data.data)
    valeur = float(data.data)
    if valeur != 0:
        frames_in[sensory_nb] = valeur

# Display time.  Must be called in the main SNN loop.
def display_chrono(start, label):
    global MAX_TIME_FRAME, first_frame, realtime_limit
    value_gtz = 0.1
    elapsed = time.time() - start
    topic_realtime.publish(float(elapsed - realtime_limit))
    txt = "time: %.2f" % (elapsed)
    rospy.loginfo(label + " " +txt)
    if elapsed > value_gtz:
        if len(time_frame) < MAX_TIME_FRAME:
```

```
            if first_frame == False:
                time_frame.append(elapsed)
            first_frame = False
        if len(time_frame) >= MAX_TIME_FRAME:
            avg = sum(time_frame) / len(time_frame)
            rospy.loginfo("Time frame average after " +str(MAX_TIME_FRAME)
+ ": " + str(avg))


# Function returning the layer index
def layer_index(layer):
    global SENSORY_LAYER, MOTOR_LAYER, INTER_LAYER
    layer_index = -1
    if layer == "sensory":
        layer_index = SENSORY_LAYER
    if layer == "inter":
        layer_index = INTER_LAYER
    if layer == "motor":
        layer_index = MOTOR_LAYER
    return layer_index

# Display the message if verbose mode
def Display(msg):
    if verbose:
        print(msg)

# Create the neurons
def Create_Neurons():
    global SENSORY_LAYER, MOTOR_LAYER, sensory_neurons, equation,
threshold_value, reset_value, refractory_value, inter_neurons,
motor_neurons
    Display("Creating SNN...")
    for x in range(0, sensory_neurons):
        frames_in.append(0.0)
    # Creation of the neurons
    for layer in range(SENSORY_LAYER,MOTOR_LAYER+1):
        # Neurons
        if layer == SENSORY_LAYER:
            neurons.append(NeuronGroup(sensory_neurons, equation,
threshold=threshold_value, reset=reset_value, refractory=refractory_value,
method='linear'))
            Display("Assigning SENSORY layer: " + str(layer))
        if layer == MOTOR_LAYER:
            neurons.append(NeuronGroup(motor_neurons, equation,
threshold=threshold_value, reset=reset_value, refractory=refractory_value,
method='linear'))
            Display("Assigning MOTOR layer: " + str(layer))
        if layer == INTER_LAYER:
            # Each layer must have at leat on neuron. If there is 0, we
change it to 1 (that won't be connected).
            if inter_neurons == 0:
                inter_neurons = 1
            neurons.append(NeuronGroup(inter_neurons, equation,
threshold=threshold_value, reset=reset_value, refractory=refractory_value,
method='linear'))
            Display("Assigning INTER layer: " + str(layer))
```

```
# Create the synapses (from xml)
def Create_Synpase():
    global tree
    # APPRENDRE-PYTHON.COM/PAGE-XML-PYTHON-XPATH
    syn_no = 0
    for neuron in tree.xpath("/SNN/layer/neuron"):
        layer = layer_index(neuron.getparent().get("type"))
        #print "Layer: " + str(layer)
        if layer != SENSORY_LAYER :
            layer_from = layer_index(neuron.get("layer"))
            layer_to = layer
            neuron_from = neuron.get("synapse")
            neuron_to = neuron.get("id")
            the_weights_str = neuron.get("weight")
            the_weights = the_weights_str.split(',')
            synapses.append(Synapses(neurons[layer_from],
neurons[layer_to], model='w: 1', on_pre='v += w', multisynaptic_index =
'synapse_number'))
            str_connect = "i="+str(neuron_from)+" j="+str(neuron_to)
            print "Synapse:  From: " + str(layer_from) + " To: " +
str(layer_to) + " condition: " + str_connect + " synapses: " +
the_weights_str
            synapses[syn_no].connect(i=eval(neuron_from),
j=eval(neuron_to))
            for i in range (0, len(the_weights)):
                #print "w: " + str(eval(the_weights[i]))
                synapses[syn_no].w[i] = eval(the_weights[i])
            syn_no += 1
    #sys.exit(1)


# Simulation of SNN
def Simulation():
    # Integrtion of each component in the network.
    Display("Integration of each component in the network.")
    stateMotor = StateMonitor(neurons[MOTOR_LAYER], 'v', record=True)
    spikeMonitor = SpikeMonitor(neurons[MOTOR_LAYER])
    net = Network(collect())
    net.add(neurons)
    net.add(synapses)

    Display("Restoring previously learned SNN...")
    net.store()

    init_frames_in()

    # Main loop.  Inifite if RUN mode.   Quit after X iteration if LEARNING
mode.
    theExit = False
    while True:
        cycle_msg = "\n"
        # Start the cycle and the timer.
        start = time.time()
        time.clock()


        # Restore initial SNN and monitors
```

```
        net.restore()

        # When the callback function has received all the input neurons,
assign those neurons to the input layer.
        frames_assignation = frames_in
        #rospy.loginfo("Assigned sensories: " + str(frames_assignation))

        # Assing sensory neurons from frames
        for k in range(0,sensory_neurons):
            neurons[SENSORY_LAYER].v[k] = frames_assignation[k]   # Only v
of the sensory neurons
            neurons[SENSORY_LAYER].I[k] = input_drive_current
            #rospy.loginfo("neuron " + str(k) + " v. " +
str(neurons[SENSORY_LAYER].v[k]))
            cycle_msg += "neuron " + str(k) + " v. " +
str(neurons[SENSORY_LAYER].v[k]) + "\n"
        init_frames_in()
        # Simulation execution
        net.run(simulation_lenght_ms)

        # Publish on the output topic
        for y in range(0, motor_neurons):
            #rospy.loginfo("Values to publish for neuron " + str(y) + " : "
+ str(len(stateMotor.v[y])))
            #rospy.loginfo("Neuron: " + str(y) + " Spikes: " +
str(spikeMonitor.num_spikes))
            # publish volts
            voltsToPublish = Float32MultiArray()
            voltsToPublish.data = stateMotor.v[y]
            topics_motor_volts[y].publish(voltsToPublish)
            # publish spikes
            nb_spikes = sum(spikeNo == y for spikeNo in spikeMonitor.i)
            topics_motor_spikes[y].publish(nb_spikes)
            cycle_msg += "Neuron: " + str(y) + " Spikes: " + str(nb_spikes)
+ "\n"
        topic_simulation_lenght.publish(simulation_lenght_int)
        #rospy.loginfo("Transmitted voltage values: "  +
str(len(stateMotor.v[0])))

        # End of the cycle
        os.system("clear")
        display_chrono(start, "LAST CYCLE DATA:")
        Display(cycle_msg)


# MAIN SSN Functions

# SNN
def SNN():
    rospy.loginfo("Starting SNN process...")
    start_scope()
    Create_Neurons()
    Create_Synpase()
    Simulation()


Assing_XML()
```

```
Display_Parameters()
# Declaring the topics
Display("Subscribe to the callbacks (input neurons)...")
for k in range(0, sensory_neurons):
    rospy.Subscriber("/"+SNNname+"_"+str(k+1)+"_snn_in", String,
callbackReceiveMsgFromTopic, k)

for k in range(0, motor_neurons):

topics_motor_volts.append(rospy.Publisher('motor_volts_'+SNNname+str(k+1),
Float32MultiArray, queue_size=1))

topics_motor_spikes.append(rospy.Publisher('motor_spikes_'+SNNname+str(k+1)
, Float32, queue_size=1))
topic_simulation_lenght = rospy.Publisher('simulation_lenght_'+SNNname,
Int16, queue_size=1)
topic_realtime = rospy.Publisher('realtime_'+SNNname, Float32,
queue_size=1)

SNN()
```

Fichier `input_converter.py`

```
#!/usr/bin/env python

'''
Filename: input_converter.py
Author: Jean-Sebastien Dessureault
Date created: 19/05/2017
Python version 2.7
'''

import rospy
import numpy as np
from std_msgs.msg import *
from sensor_msgs.msg import *
import string
import csv

# Registering node to ROS
node = "node_converter_input"
rospy.init_node(node, anonymous=True)
rospy.loginfo(node)

# Retriving parameters from launcher
verbose = rospy.get_param("~verbose")
SNNname = rospy.get_param("~SNNname")
pathSNN = rospy.get_param("~path")
topics_to_convert = rospy.get_param("~topics_to_convert")
p_topic = []
p_type = []
p_field = []
```

```
p_min = []
p_max = []
p_stringfile = []
out_topic = []


for x in range (0, topics_to_convert):
    p_topic.append(rospy.get_param("~input_topic_"+str(x+1)))
    p_type.append(rospy.get_param("~topic_type_"+str(x+1)))
    p_field.append(rospy.get_param("~input_field_"+str(x+1)))
    p_min.append(rospy.get_param("~input_min_"+str(x+1)))
    p_max.append(rospy.get_param("~input_max_"+str(x+1)))
    p_stringfile.append(rospy.get_param("~stringfile_"+str(x+1)))

def MinMax(value_to_normalize, min, max):
    normalized_value = float((value_to_normalize - min) / (max - min))
    return normalized_value

def callback(data, neuron_nb):
    #rospy.loginfo(data)
    field = eval("data"+p_field[neuron_nb])
    if (p_stringfile[neuron_nb] == ""):
        # Numeric field
        converted_value = float(field)
        converted_value = MinMax(converted_value, p_min[neuron_nb],
p_max[neuron_nb])
        #rospy.loginfo("Le callback publie la valeur %f sur le topic.",
converted_value)
        out_topic[neuron_nb].publish(str(converted_value))
        #print "Publie: " + str(converted_value) + " sur " + str(neuron_nb)
    else:
        # string field
        csv_file = pathSNN + "csv/" + p_stringfile[neuron_nb]
        #print csv_file
        with open(csv_file, 'r') as f:
            reader = csv.DictReader(f, delimiter=",", quotechar="|")
            try:
                for row in reader:
                    if (row['string'] == str(field)):
                       #print "publish: " + str(row['voltage'])
                        out_topic[neuron_nb].publish(str(row['voltage']))
            except csv.Error as e:
                 rospy.logerror("A string received in the converter
parameter (input_parameter.py) has no match in the specified csv file.
Giving 0.0 value.")
                 out_topic[neuron_nb].publish("0.0")

if verbose:
    rospy.loginfo("Mapping and converting topics for sensory inputs of the
SNN.  Do NOT interrupt while SNN execute. ")

for x in range (0, topics_to_convert):
    rospy.Subscriber(p_topic[x], eval(p_type[x]) , callback, x)
    out_topic.append(rospy.Publisher('/'+SNNname+'_'+str(x+1) + "_snn_in",
String, queue_size=1))
    #print "Declaration du topic: " + '/ssn_in_'+SNNname+'_'+str(x+1)
```

```
rospy.spin()
```

Fichier `plot_volts.py`

```
#!/usr/bin/env python
import rospy
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import sys
from std_msgs.msg import Float32, Float32MultiArray, Int16
from lxml import etree

node = "node_volt_plot"
rospy.init_node(node, anonymous=True)
rospy.loginfo(node)

SNNname = rospy.get_param("~SNNname")
pathSNN = rospy.get_param("~path")
xml = rospy.get_param("~xml")

simulation_length = 1

x_lim = 1
old_x_lim = x_lim

volts = []
times = []
iTime = []
topics_motor = []
plot_array = []

sensory_neurons = 0
inter_neurons = 0
motor_neurons = 0
inter_layers = 0

fig1 = plt.figure()
ax1 = fig1.add_subplot(111)

title = "MOTOR voltage for "+SNNname
plt.title(title)
plt.xlabel("Frames")
plt.ylim(-1.2, 1.2)
plt.ylabel("volts")
plt.xlim(0,x_lim)
plt.grid(True)
#legend = plt.legend()


def count_neurons():
    global tree, sensory_neurons, inter_neurons, motor_neurons,
inter_layers
    # Defining number of each layer
    for neuron in tree.xpath("/SNN/layer/neuron"):
```

```
        layer_type = neuron.getparent().get("type")
        if layer_type == "sensory":
            sensory_neurons+=1
        if layer_type == "inter":
            inter_neurons+=1
            inter_layers = 1
        if layer_type == "motor":
            motor_neurons+=1

def init_iTime():
        for neuron_nb in range (0,motor_neurons):
                iTime.append(0)

def init_volts_and_times():
        for neuron_nb in range (0,motor_neurons):
                volts.append([])
                times.append([])

def callbackVolts(data, neuron_nb):
        del times[neuron_nb][:]
        del volts[neuron_nb][:]
        global x_lim, old_x_lim, fig1, simulation_length

        # Ajust the limit.
        x_lim = len(data.data)

        # If the limit has changed, modify ploting scale (x_limit)
        #print "x_lim: " + str(x_lim) + " " + "old_x_lim: " + str(old_x_lim)
        if x_lim != old_x_lim:
                old_x_lim = x_lim
                plt.xlim(0,x_lim)

        # Assign voltage
        for j in range(0,len(data.data)):
                volts[neuron_nb].append(data.data[j])

        # Assing time and divide by simulation lenght
        times[neuron_nb].append(range(0,len(volts[neuron_nb])))
        #for j in range(0, len(data.data)):
        #       times[neuron_nb][0][j] /= simulation_lenght



def callbackSimulationLenght(data):
        global simulation_length, fig1
        simulation_length = data.data
        plt.title(title + " Simulation length: " + str(simulation_length))


xml_file = pathSNN + "xml/" + xml
rospy.loginfo("xml:" + xml_file)
try:
    tree = etree.parse(xml_file)
    count_neurons()
except:
    rospy.loginfo("Error XML file: " + xml_file)
    sys.exit(1)
```

```
# Initialize arrays
init_volts_and_times()

for neuron_nb in range (0,motor_neurons):
        topics_motor.append(rospy.Subscriber('/motor_volts_'+SNNname+str(neu
ron_nb+1), Float32MultiArray, callbackVolts, neuron_nb))

topic_simulation_lenght =
rospy.Subscriber('/topic_simulation_lenght_'+SNNname, Int16,
callbackSimulationLenght)

def update_line(num, data, ax):

        # Get the smaller lenght of the arrays, and use only those data to
plot.  If not: xdata and ydata must be the same (error).
        smaller = 999999
        for neuron_nb in range (0, motor_neurons):
                if len(volts[neuron_nb]) < smaller:
                        smaller = len(volts[neuron_nb])

        for neuron_nb in range (0, motor_neurons):
                #rospy.loginfo("Upadate frame: " +
str(len(times[neuron_nb])) + " " + str(len(volts[neuron_nb])))

                plot_array[neuron_nb].set_data(times[neuron_nb][:smaller],
volts[neuron_nb][:smaller])
                plot_array[neuron_nb].set_label("Neuron: " +
str(neuron_nb+1))
                legend = plt.legend()
        return ax,



# Continue defining graphic
for neuron_nb in range(0, motor_neurons):
        plot_tmp, = ax1.plot(times[neuron_nb], volts[neuron_nb])
        plot_array.append(plot_tmp)

# Defining animation
# Animation parameters
# - figure to plot
# - update function
# - number of frames to cache
# - interval. Delay between frames in ms. (def: 200)
# - repeat: loop animation
# - blit: quality of the animation
simulation = animation.FuncAnimation(fig1, update_line, 25,
fargs=(volts,ax1), interval=50, repeat=True)
#simulation.save(filename='animation.mp4', fps=30, dpi=300)
plt.show()

#rospy.spin()
```

Fichier `plot_input.py`

```
#!/usr/bin/env python
import rospy
import numpy as np
import sys
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from std_msgs.msg import Float32, String, Float32MultiArray, Int16
from lxml import etree

node = "node_input_plot"
rospy.init_node(node, anonymous=True)
rospy.loginfo(node)

SNNname = rospy.get_param("~SNNname")
pathSNN = rospy.get_param("~path")
xml = rospy.get_param("~xml")

simulation_length = 1000
x_lim = simulation_length

volts = []
times = []

#topics_input = []
plot_array = []

sensory_neurons = 0
inter_neurons = 0
motor_neurons = 0
inter_layers = 0

fig1 = plt.figure()
ax1 = fig1.add_subplot(111)

title = "SENSORY voltage for "+SNNname
plt.title(title)
plt.xlabel("Frames")
plt.ylim(0, 1.1)
plt.ylabel("volts")
plt.xlim(0,x_lim)
plt.grid(True)

def count_neurons():
    global tree, sensory_neurons, inter_neurons, motor_neurons,
inter_layers
    # Defining number of each layer
    for neuron in tree.xpath("/SNN/layer/neuron"):
        layer_type = neuron.getparent().get("type")
        if layer_type == "sensory":
            sensory_neurons+=1
        if layer_type == "inter":
            inter_neurons+=1
            inter_layers = 1
        if layer_type == "motor":
```

```
            motor_neurons+=1

def init_volts_and_times():
        for neuron_nb in range (0,sensory_neurons):
                volts.append([])
                times.append([])


def reinit_if_needed(neuron_nb):
        global simulation_length
        if len(volts[neuron_nb]) >= simulation_length:
                del times[neuron_nb][:]
                del volts[neuron_nb][:]


def callback_input(data, neuron_nb):
        global simulation_length
        value = float(data.data)
        #print "CALLBACK INPUT: " + str(value)
        reinit_if_needed(neuron_nb)
        volts[neuron_nb].append(value)
        times[neuron_nb].append(len(volts[neuron_nb]))


xml_file = pathSNN + "xml/" + xml
rospy.loginfo("xml:" + xml_file)
try:
    tree = etree.parse(xml_file)
    count_neurons()
except:
    rospy.loginfo("Error XML file: " + xml_file)
    sys.exit(1)

# Initialize arrays
init_volts_and_times()

for neuron_nb in range (0,sensory_neurons):
        rospy.loginfo('Subscribe to
/'+SNNname+'_'+str(neuron_nb+1)+'_snn_in')
        #topics_input.append(rospy.Subscriber('/'+SNNname+'_'+str(neuron_nb+
1)+'_snn_in', String, callbackVolts, neuron_nb))
         rospy.Subscriber("/"+SNNname+"_"+str(neuron_nb+1)+"_snn_in",
String, callback_input, neuron_nb)

def update_line(num, data, ax):
        global volts, times
        #print volts[neuron_nb]
        # Get the smaller lenght of the arrays, and use only those data to
plot.  If not: xdata and ydata must be the same (error).
        smaller = 999999
        for neuron_nb in range (0, sensory_neurons):
                if len(volts[neuron_nb]) < smaller:
                        smaller = len(volts[neuron_nb])

        for neuron_nb in range (0, sensory_neurons):
                #rospy.loginfo("Upadate frame: " +
str(len(times[neuron_nb])) + " " + str(len(volts[neuron_nb])))
```

```
                plot_array[neuron_nb].set_data(times[neuron_nb][:smaller],
volts[neuron_nb][:smaller])
                plot_array[neuron_nb].set_label("Neuron: " +
str(neuron_nb+1))
                legend = plt.legend()

        return ax ,


# Continue defining graphic
for neuron_nb in range(0, sensory_neurons):
        plot_tmp, = ax1.plot(times[neuron_nb], volts[neuron_nb])
        plot_array.append(plot_tmp)

# Defining animation
# Animation parameters
# - figure to plot
# - update function
# - number of frames to cache
# - interval. Delay between frames in ms. (def: 200)
# - repeat: loop animation
# - blit: quality of the animation
simulation = animation.FuncAnimation(fig1, update_line, 25,
fargs=(volts,ax1), interval=50, repeat=True)
#simulation.save(filename='animation.mp4', fps=30, dpi=300)
plt.show()

rospy.spin()
```

Fichier `plot_spikes.py`

```
#!/usr/bin/env python
import rospy
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from std_msgs.msg import Float32
import sys
from lxml import etree

node = "node_spikes_plot"
rospy.init_node(node, anonymous=True)
rospy.loginfo(node)

pathSNN = rospy.get_param("~path")
xml = rospy.get_param("~xml")
SNNname = rospy.get_param("~SNNname")

x_lim = 50
y_lim = 5
nb_spikes = []
times = []
iTime = []
topics_motor_spikes = []
plot_array = []
```

```
sensory_neurons = 0
inter_neurons = 0
motor_neurons = 0
inter_layers = 0

fig1 = plt.figure()
ax1 = fig1.add_subplot(111)

plt.title("Spikes for "+SNNname)
plt.ylim(0, y_lim)
plt.ylabel("Spikes")
plt.xlabel("Simulations through time")
plt.xlim(0,x_lim)
plt.grid(True)
#legend = plt.legend()

def count_neurons():
    global tree, sensory_neurons, inter_neurons, motor_neurons,
inter_layers
    # Defining number of each layer
    for neuron in tree.xpath("/SNN/layer/neuron"):
        layer_type = neuron.getparent().get("type")
        if layer_type == "sensory":
            sensory_neurons+=1
        if layer_type == "inter":
            inter_neurons+=1
            inter_layers = 1
        if layer_type == "motor":
            motor_neurons+=1


def init_iTime():
        for neuron_nb in range (0,motor_neurons):
                iTime.append(0)

def init_spikes_and_times():
        for neuron_nb in range (0,motor_neurons):
                nb_spikes.append([])
                times.append([])

def callbackNbSpike(data, neuron_nb):
        if iTime[neuron_nb] >= x_lim:
                del times[neuron_nb][:]
                del nb_spikes[neuron_nb][:]
                iTime[neuron_nb] = 0
        nb_spikes[neuron_nb].append(data.data)
        times[neuron_nb].append(iTime[neuron_nb])
        iTime[neuron_nb] += 1


xml_file = pathSNN + "xml/" + xml
rospy.loginfo("xml:" + xml_file)
try:
    tree = etree.parse(xml_file)
    count_neurons()
except:
```

```
    rospy.loginfo("Error XML file: " + xml_file)
    sys.exit(1)


# Initialize arrays
init_iTime()
init_spikes_and_times()

for neuron_nb in range (0,motor_neurons):
        topics_motor_spikes.append(rospy.Subscriber('/motor_spikes_'+SNNname
+str(neuron_nb+1), Float32, callbackNbSpike, neuron_nb))

def update_line(num, data, ax):
        # Get the smaller lenght of the arrays, and use only those data to
plot.  If not: xdata and ydata must be the same (error).
        smaller = 999
        for neuron_nb in range (0, motor_neurons):
                if len(nb_spikes[neuron_nb]) < smaller:
                        smaller = len(nb_spikes[neuron_nb])

        for neuron_nb in range (0, motor_neurons):
                #rospy.loginfo("Upadate frame: " +
str(len(times[neuron_nb])) + " " + str(len(volts[neuron_nb])))

                plot_array[neuron_nb].set_data(times[neuron_nb][:smaller],
nb_spikes[neuron_nb][:smaller])
                plot_array[neuron_nb].set_label("Neuron: " +
str(neuron_nb+1))
                legend = plt.legend()
        return ax,


# Continue defining graphic
for neuron_nb in range(0, motor_neurons):
        plot_tmp, = ax1.plot(times[neuron_nb], nb_spikes[neuron_nb])
        plot_array.append(plot_tmp)

# Defining animation
# Animation parameters
# - figure to plot
# - update function
# - number of frames to cache
# - interval. Delay between frames in ms. (def: 200)
# - repeat: loop animation
# - blit: quality of the animation
simulation = animation.FuncAnimation(fig1, update_line, 25,
fargs=(nb_spikes,ax1), interval=50, repeat=True)
#simulation.save(filename='animation.mp4', fps=30, dpi=300)
plt.show()

#rospy.spin()
```

Fichier `plot_connectivity.py`

```
#!/usr/bin/env python

'''
Filename: SNN.py
Author: Jean-Sebastien Dessureault
Date created: 01/06/2016
Python version 2.7
'''
from brian2 import *
prefs.codegen.target = "cython"

import sys
import rospy
import numpy as np
import time
import string
from lxml import etree
import matplotlib.pyplot as plt
import random as rnd

# Registering node to ROS
node = "node_connectivity_plot"
rospy.init_node(node, anonymous=True)
rospy.loginfo(node)

# Retriving parameters from launcher
pathSNN = rospy.get_param("~path")
xml = rospy.get_param("~xml")
SNNname = rospy.get_param("~SNNname")

verbose = True

sensory_neurons = 0
motor_neurons = 0
inter_neurons = 0
inter_layers = 0
synapse_delay = ""
input_drive_current = 0
tau = 0* ms
threshold_value = 0
refractory_value = 0* ms
reset_value = 0
simulation_lenght_int = 0

# Displaying parameters to console
rospy.loginfo("----Parameters received from launcher ----")
rospy.loginfo("SNNname: " + SNNname)
rospy.loginfo("xml: " + xml)
rospy.loginfo("verbose: " + str(verbose))
rospy.loginfo("path: " + pathSNN)


# Registering node to ROS
#rospy.init_node('node_plot_connectivity_'+SNNname, anonymous=True)
```

```
#rospy.loginfo("SNN - Plot architecture - " + SNNname)


# Function returning the layer index
def layer_index(layer):
    global SENSORY_LAYER, MOTOR_LAYER
    INTER_LAYER = 1
    layer_index = -1
    if layer == "sensory":
        layer_index = SENSORY_LAYER
    if layer == "inter":
        layer_index = INTER_LAYER
    if layer == "motor":
        layer_index = MOTOR_LAYER
    return layer_index

def nb_neuron(layer_no):
    global sensory_neurons, inter_neurons, motor_neurons
    if layer_no == 0:
        return sensory_neurons
    if layer_no == 1:
        return inter_neurons
    if layer_no == 2:
        return motor_neurons
    return -1

# Function return a certain offset for the inter neurons
def offset(layer, neuron):
    global INTER_LAYER
    if layer == INTER_LAYER:
        maxi = float(nb_neuron(layer))
        middle = float(maxi / 2.0)
        current = float(neuron)
        value = float(abs(current - middle) / middle)
        value = pow(value,2)
        value = value - 0.5
        print "Offset: " + str(value)
        return value
    return 0.0

# Define neurons positions in the graph
neurons_x = []
neurons_y = []
neurons_names = []
synapse_layer_from_to = []
synapse_neuron_from_to = []
synapse_weight = []

if xml == "":
    rospy.loginfo("No xml definition to plot... ")
    rospy.spin()


xml_file = pathSNN + "xml/" + xml
rospy.loginfo("xml:" + xml_file)

try:
```

```
    tree = etree.parse(xml_file)
    # Get the SNN variables.  Will be placed in a legend beside the graph.
    for rnd in tree.xpath("/SNN"):
        print "Look for variables"
        name = rnd.get("name")
        synapse_delay = rnd.get("synapse_delay")
        input_drive_current = float(rnd.get("input_drive_current"))
        tau = int(rnd.get("tau")) *ms
        threshold_value = rnd.get("threshold")
        reset_value = rnd.get("reset")
        refractory_value = int(rnd.get("refractory")) * ms
        simulation_lenght_int = float(rnd.get("sim_lenght"))

    # Defining number of each layer
    sensory_neurons = 0
    inter_neurons = 0
    motor_neurons = 0
    inter_layers = 0
    for neuron in tree.xpath("/SNN/layer/neuron"):
        layer_type = neuron.getparent().get("type")
        if layer_type == "sensory":
            sensory_neurons+=1
        if layer_type == "inter":
            inter_neurons+=1
            inter_layers = 1
        if layer_type == "motor":
            motor_neurons+=1
        neurons_names.append(neuron.text)

    # Constants
    SENSORY_LAYER = 0
    INTER_LAYER = 1
    MOTOR_LAYER = 2

    rospy.loginfo("Sensory neurons " + str(sensory_neurons))
    rospy.loginfo("Inter layers " + str(inter_layers))
    rospy.loginfo("Inter neurons " + str(inter_neurons))
    rospy.loginfo("Motor neurons " + str(motor_neurons))

    for layer_no in range(SENSORY_LAYER,MOTOR_LAYER+2):
        #print "Processing layer #" + str(layer_no)
        for neuron_no in range(0, nb_neuron(layer_no)):
            #print "...Processing neuron #" + str(neuron_no) + " of layer
#" + str(layer_no)
            neurons_y.append(layer_no+offset(layer_no, neuron_no))
            neurons_x.append(neuron_no)

    # Define synapses links between the neurons
    nb_synapses = 0
    for neuron in tree.xpath("/SNN/layer/neuron"):
        layer_type_to = neuron.getparent().get("type")
        neuron_id = neuron.get("id")
        synapse_id = neuron.get("synapse")
        if synapse_id != None:
            synapse_layer = neuron.get("layer")
            str_tmp = neuron.get("weight")
```

```
            synapse_weight_tmp = str_tmp.split(",")
            i = 0
            for syn in synapse_id.split(","):
                print "...Processing synapse #" + str(nb_synapses)
                sl = layer_index(synapse_layer)
                nl = layer_index(layer_type_to)
                synapse_layer_from_to.append((sl+offset(sl,int(syn)),
nl+offset(nl, int(neuron_id))))
                synapse_neuron_from_to.append((int(syn), int(neuron_id)))
                synapse_weight.append(synapse_weight_tmp[i])
                #print "nb_synapses: " + str(nb_synapses) + " W: " +
str(synapse_weight[nb_synapses])
                i += 1
                nb_synapses += 1

except:
    rospy.loginfo("Error parsing XML file: " + xml_file)
    sys.exit(1)

plt.scatter(neurons_y, neurons_x, c='black', s=512, marker='o')
dy_annotation = 0.25
for i, label in enumerate(neurons_names):
    plt.annotate(label, (neurons_y[i], neurons_x[i]), xytext=(neurons_y[i],
neurons_x[i]+dy_annotation))

#print synapse_layer_from_to
#print synapse_neuron_from_to

for i in range(0, nb_synapses):
    plt.plot(synapse_layer_from_to[i], synapse_neuron_from_to[i],
linestyle='solid', color='black')
    de_x = synapse_layer_from_to[i][0]
    de_y = synapse_neuron_from_to[i][0]
    a_x = synapse_layer_from_to[i][1]
    a_y = synapse_neuron_from_to[i][1]
    #print "De: x: " + str(de_x) + " y: " + str(de_y) + "  A: x: " +
str(a_x) + " y: " + str(a_y)
    px =  (de_x + a_x) / 2.0
    py =  (de_y + a_y) / 2.0
    #print str(i) + " y: " +str(py) + " x: " + str(px) + " w: " +
str(synapse_weight[i])
    plt.annotate('W: ' +str(synapse_weight[i]), xy=(px,py) ,
xytext=(px,py))

#plt.title("Architecture - " + name)
#plt.margins(y=0.8)
plt.subplots_adjust(top=0.75)
plt.suptitle(
    "Architecture - " + name +
    "\nInput drive current: " + str(input_drive_current)  + " volts" +
    "\nThreshold: " + str(threshold_value)  + " volts" +
    "\nTau: " + str(tau)  +
    "\nRefractory: " + str(refractory_value) +
    "\nReset: " + str(reset_value) + " ms" +
    "\nSimulation lenght: " + str(simulation_lenght_int) + " ms",
    ha='center', fontsize=8)
```

```
# Remove ticks (axis scaling) and axis
plt.xticks([],[])
plt.yticks([],[])
plt.axis('off')
# Show the graph
plt.show()
```